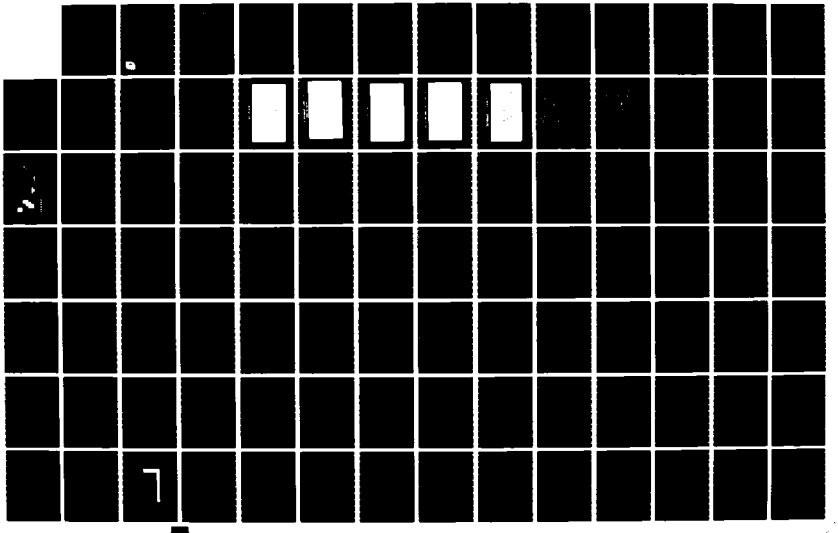
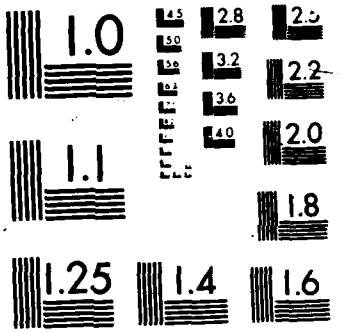
AD-A172 747 PROCEEDINGS OF THE IDA (INSTITUTE FOR DEFENSE ANALYSES) 1/4
WORKSHOP ON FORMA.. (U) INSTITUTE FOR DEFENSE ANALYSES
ALEXANDRIA VA W T MAYFIELD ET AL. NOV 85 IDA-M-135
UNCLASSIFIED DECL IDAH030579 IDA/HQ-85-30579 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

(2)

IDA MEMORANDUM REPORT M-135

PROCEEDINGS OF THE SECOND IDA WORKSHOP ON FORMAL SPECIFICATION AND VERIFICATION OF Ada* JULY 23-25, 1985

W. T. Mayfield
S. R. Welke

November 1985

Prepared for
Office of the Under Secretary of Defense for Research and Engineering

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

IDA Log No. HQ 85-30579

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE AD-A172 741

1a REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Public release; distribution unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) M-135	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses 6b OFFICE SYMBOL CSED	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22305	7b ADDRESS (City, State, and Zip Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION Ada Joint Program Office 8b OFFICE SYMBOL (If applicable) AJPO	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) 1211 Fern St., C107 Arlington, VA 22202	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-4-263 / WORK UNIT ACCESSION NO.
11 TITLE (Include Security Classification) Proceedings of the Second IDA Workshop on Formal Specification and Verification of Ada, July 23-25, 1985 (U)	
12 PERSONAL AUTHOR(S) W.T. Mayfield, S.R. Welke	
13a TYPE OF REPORT Final 13b TIME COVERED FROM _____ TO _____ 14 DATE OF REPORT (Year, Month, Day) 1985 November 15 PAGE COUNT 346	
16 SUPPLEMENTARY NOTATION	
17 COSATI CODES FIELD / GROUP / SUB-GROUP	18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Ada, verification, specification, secure systems, semantic, concurrency, computer security, software, support library, run-time support library

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The Second Workshop identified current issues in Ada Verification and focused on what is needed to build the foundation of an Ada Verification Techology. IDA workshops will continue to be a meeting place for accessing the current state-of-the-art, identifying promising research areas, monitoring ongoing verification work, promoting the use of the evolving technology, and ensuring that valuable outputs from one area are fed into other areas. The desired product of these workshops will be recommendations to various bodies to coordinate and sponsor certain R&D activities. Working groups on special topics were also established.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

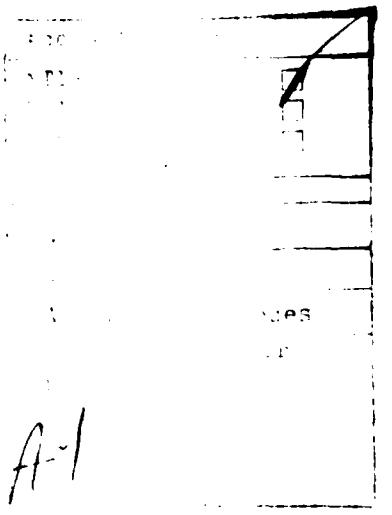
SECURITY CLASSIFICATION OF THIS PAGE

IDA MEMORANDUM REPORT M-135

PROCEEDINGS OF THE SECOND IDA WORKSHOP ON FORMAL SPECIFICATION AND VERIFICATION OF Ada* JULY 23-25, 1985

W. T. Mayfield
S. R. Welke

November 1985



A-1

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-4-263

Foreword

These Proceedings of the Second Workshop on Formal Specification and Verification of Ada, held at the Institute for Defense Analyses (IDA), are composed in part of papers and slides supplied by the speakers, and in part of summaries of the talks and discussions edited from notes taken during the Workshop.

The purpose of this second two-day workshop was to continue discussions on issues raised in the initial workshop held in March 1985, to further identify current issues in Ada verification, and to focus on what is needed to build the foundations of an Ada Verification Technology.

At the end of the first workshop, several conclusions were reached. First, there was general agreement that R&D over the past several years has yielded some useful techniques. Second, the participants determined that these IDA Workshops would serve as a meeting place where a group of experts could assess the current state-of-the-art, identify promising research areas, monitor ongoing verification work, promote the use of the evolving technology, and ensure that valuable outputs from one area were fed into other areas. Lastly, the participants decided that the desired product of these workshops would be recommendations to various bodies to coordinate and sponsor certain R&D activities.

In an attempt to foster results from those attending these workshops, working groups on special topics were established. It was envisaged that the groups would prepare material for the next workshop and, where appropriate, draft their recommendations to be forwarded to the relevant official bodies after discussion at that meeting. Working groups were formed under the topics shown below.

SECURE SYSTEMS chaired by M. Zuk, MITRE Corporation

NEAR TERM VERIFICATION chaired by J. McHugh, Research Triangle Institute

FORMAL SEMANTICS AND CONCURRENCY chaired by N. Cohen, SofTech, Inc.

SPECIFICATION LANGUAGES chaired by F. von Henke, SRI International

VERIFICATION IN LIFE CYCLES chaired by A. Marmor-Squires, TRW, Defense Systems Group

"OFFICIAL" CLUSTERS chaired by R. Platek, Odyssey Research Associates, Inc.

As the time for convening the second workshop drew closer, it became apparent that the above topics were really focal areas rather than actual working groups. Interest in the different groups was so imbalanced that there seemed a need to combine some of them. At the same time, it became apparent that the majority of the prospective participants wanted to attend all sessions rather than being restricted to one working group. Thus, Clyde Roby as General Chair, in concert with the working group chairs, revamped the format for the second workshop to allow plenary sessions for all presentations and general discussions. By the end of the second workshop, two new interest groups were formed to replace the old working groups - SECURE SYSTEMS chaired by Margie Zuk and Richard Platek, and FORMAL SPECIFICATION AND SEMANTICS chaired by Norm Cohen and Friedrich von Henke.

The workshop was opened Tuesday afternoon by Clyde Roby, who welcomed all the participants and announced the change in format from separate working groups to plenary sessions. The program began with introductory talks given by Paul Cohen of the Ada Joint Program Office (AJPO), John Faust of the Rome Air Development Center (RADC), and Col. Joseph Greene of the DoD Computer Security Center (DODCSC).

Paul Cohen stressed the importance that the AJPO places on the development of Ada verification technology and confirmed that the AJPO supports the efforts of this group.

John Faust followed with what he thought should be the goals of these workshops. These goals included establishing and nurturing an Ada verification peer review group, identifying the state-of-the-art of verification, recommending technical directions for Ada verification, and coordinating Ada verification with other agencies.

Col. Greene focused on the need for Ada verification to support computer security, citing the lag in achieving computer security as compared to communications security. He indicated that President Reagan's National Security Decision Directive 145 emphasizes the need for computer security R&D as part of a national program to improve the security posture of Automated Information Systems. Col. Greene then discussed the near-term (5 yrs) and longer-term (15 yrs) goals within DoD to deploy trusted systems and to achieve interoperability of systems. Placing Ada and Verification in perspective, Col. Greene discussed the importance of both to the DoD program. Ada is important because it is the chosen language for mission critical software for secure systems. Verification is important because it will give us additional assurances as to the trustworthiness of a "trusted" computer base.

The technical program began with Ann Marmor-Squires presenting the charter of her working group and the key issues concerning the role of verification in the "Life Cycle." These issues included defining the life cycle, determining the cost of performing verification, identifying the role of automated tools, and establishing how to begin integrating verification into the life cycle.

Ann was followed by Karl Nyberg. Karl, standing in for John McHugh, presented the focus of the Near Term working group. This focus was on the adaptation of existing languages, tools, and methods to provide for formal specification and verification of Ada. Issues included the potential for language changes in 1988, and the need for Ada formal semantics before Ada verification systems can be built. There was an additional speaker from this group. Tom Kraly, of IBM, spoke informally on the "Clean Room" approach, which is based on the work of Harlan Mills. In this approach, semi-formal, manual methods are used during the development process to avoid the introduction of errors from the beginning.

The next speaker, Friedrich von Henke, discussed the role of specification languages in verification. He presented the charter of his working group and highlighted topics which must be addressed. These topics included how to specify concurrency and real-time properties, possibilities for an Ada Specification Language, and the requirements on a specification language. Norm Cohen completed the Tuesday afternoon session with a proposal for a "conservative" implementation of Ada as a way to simplify Ada semantics.

The Wednesday morning session began with David Luckham's proposal for Ada formal semantics that included the concept of two semantics; one, an "instrumented" compiler (capable of explaining what it is doing when queried by a user) and the other, an axiomatic proof system. This stimulating proposal evoked a lengthy discussion.

David was followed by Kurt Hansen of Dansk Datamatik Center who was invited to speak by the Formal Semantics Working Group. Kurt presented the European project to develop a formal definition of Ada and provided drafts of several documents on the project to the workshop participants. Copies of most of the documents can be found Appendix B. Certain papers were not available for release, reproduction, and inclusion herein.

The morning session was completed by Norm Cohen, who presented a notation that is a variation on Dijkstra's notation and has particular advantages for Ada proof rules.

Margie Zuk kicked off the Wednesday afternoon session by presenting both the areas of concern and the goals of the Secure Systems working group. The features of the Ada Language that create concern about the design of secure systems include language constructs, run-time support libraries, and the issue of compiler unpredictability. The goals include better delineation of the features of Ada which introduce security concerns, study of the "conservative" compiler introduced by Norm Cohen, and determination of the language restrictions necessary for secure systems. Margie invited two additional speakers to discuss Ada Run-time Support Libraries. Juern Jurgens from Softech and Omar Ahmed from Verdix each outlined the key features of their companies' run-time support libraries.

The final day consisted of summaries by working group chairs and recommendations for actions to be taken in the area of formal verification of Ada. These recommendations included:

a. Developing several formal semantics for Ada

b. Developing a "conservative" compiler and an "instrumented" compiler

c. Experimenting with specifying programs in ANNA

d. Performing basic research in specifying concurrency, real-time behavior, and floating point arithmetic

e. Developing "Ada oriented" requirements, designs, and specification languages

f. Determining restrictions on Ada so that it can be used for security

g. Studying the security issues of Ada Run-Time Support Libraries (RSLs)

h. Identifying and tracking ongoing efforts in secure Ada systems

John McHugh proposed four near-term efforts. These were:

a. Prototype development

b. Investigation of semi-formal methods

c. Identification of Ada-specific verification problems

d. Identification of constraints on run-time support and code generation

ACKNOWLEDGMENT

The Institute for Defense Analyses would like to thank all the Working Group Chairs and, in particular, Richard Platek and his staff at Odyssey Research Associates, Inc., for their assistance in the preparation of these proceedings.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TERMS AND ABBREVIATIONS

ACVC	Ada Compiler Validation Critieria
AIS	Automated Information Systems
AISS	Automated Information Systems Security
AJPO	Ada Joint Program Office
ANSI	American National Standards Institute
ASOS	Army Secure Operating System
CCITT	Consultative Committee on International Telephone and Telegraph (Comite Consultatif International Telephonique et Telegraphique)
CEC	Commission of the European Communities
CM	Configuration Management
DAC	Discretionary Access Control
DBML	Database Manipulation Language
DDC	Dansk Datamatik Center
DML	Data Manipulation Language
DOC	Documentation
DoD	Department of Defense
DODCSC	DoD Computer Security Center
DTLS	Descriptive Top Level Specifications
ESPRIT	European Strategic Programme for Research and Development of Information Technologies
FD	Formal Definition
FTLS	Formal top Level Specifications
GKS	Graphics Kernel System
IDA	Institute for Defense Analyses
I/O	Input/Output
LRM	Language Reference Manual (ANSI/MIL-STD-1815A)
MAC	Mandatory Access Control
MAP	Multi-Annual Programme
MIL-STD	Military Standard
MLS	Mid-Level Specification
NSDD	National Security Decision Directive
OB	Orange Book
PC	Personal Computer
PDL	Program Design Language
PHIGS	The Progammer's Hierarhical Interactive Graphics Standards
R&D	Research & Development
RADC	Rome Air Development Center
RSL	Run-Time Support Library

SDI	Strategic Defense Initiative
SETL	Set Theoretic Language
SFD	Static Frame Descriptor
SIGAda	Special Interest Group on Ada (ACM)
SMoLCS	Structured Monitored Linear Concurrent Systems
TCB	Trusted Computing Base
TDB	Trusted Database
TNB	Trusted Network Base
VDM	Vienna Development Method

1 TUESDAY AFTERNOON SESSION

1.1 Introductory Talks

The Workshop began with several short introductory talks. Clyde Roby of the Institute for Defense Analyses (IDA) opened the Workshop, and announced that a decision had been made to change the Workshop's format. Originally, the six working groups formed at the end of the first workshop were going to meet in parallel sessions. However, once the working group chairs got together, they decided to have their groups meet serially so that everyone could attend every groups's talks. Mr. Roby also also announced that an account (ADA-INFORMATION, password Ada) had been created at USC-ECLB to serve as a clearing-house for Ada-related activities.

Next, Paul Cohen of the Ada Joint Program Office (AJPO) briefly described the AJPO. The five principal thrusts of the AJPO are shown below:

a. Standards

b. Education and Training

c. Validation

d. Environments

e. Trusted Software and Verification

All of these efforts are heavily sponsored at the AJPO. Mr. Cohen also mentioned that he is excited to see so much interest in Ada verification because so little has been done in the area.

The next speaker was John Faust of the Rome Air Development Center (RADC). He listed several goals of IDA's Ada verification effort:

a. To establish an Ada verification peer review group

b. To identify the state of the art in Ada verification

c. To recommend technical directions for Ada verification

d. To coordinate Ada verification with other agencies (e.g., the AJPO and STARS Program Office)

The IDA effort should highlight computer security concerns, but should not be limited to security. The effort should also include the verification of properties other than access control. Verification of both design ("A1" verification) and code ("beyond A1" verification) should be addressed.

1.2 Why the DoD Computer Center (DODCSC) is Interested in Ada - Col. Joseph Greene, DODCSC

Telecommunications security and Automated Information Systems Security (AISS) are converging. However, the two have different levels of maturity. In telecommunications security, we have the new technology; it is primarily a matter of getting it distributed. AISS technology is about 10-15 years behind telecommunications security, so there is a need for research and development.

This convergence is recognized in the President's National Security Decision Directive 145 (NSDD-145). In response to NSDD-145, the DoD has formulated 5- and 14-year goals for AIS. The 15-year goal is to establish interoperability within the DoD. The 5-year, mid-term goal is to deploy trustable automated information systems using Common Ada Program Support Environment (APSE) Interface Sets (CAIS's). To accomplish the way industry designs and builds word processors, PCs, minicomputers, mainframes, database management systems, local area networks and network components, and multimedia systems. There is a 15-year commitment to create a new technology base and distribute it to industry.

The DoD Trusted Computer System Evaluation Criteria (CSC-STD-001-83; a.k.a. the "Orange Book") defines certain fundamental requirements for AISS. These requirements include (at various levels of trustedness) a security policy, accountability (auditing), certain assurance methods, and requirements for trusted configuration management (CM) and trusted distribution. The C Division of systems primarily addresses discretionary access control (DAC). Systems in the C Division are subject to so-called "Trojan Horse" attacks. Higher divisions (B and A) address mandatory access control (MAC) which involves controlling access to data labelled with National Security classifications. In these higher Divisions, the Trojan Horse threat is countered by more rigorous assurance methods (including formal verification for A Division) and rigid configuration control.

There is a trade-off in near-term funding between formal verification and CM technology. The technology base for formal verification is at present very thin.

Ada comes in because it will be used for mission-critical software in security systems. It is also an avenue to distribute trusted system technology to the computer industry. The DODCSC supports the following policy:

a. Use and support Ada standards.

b. Monitor and incorporate emerging standards.

c. Code entirely in machine-independent Ada.

d. Use Ada syntax and semantics to the maximum extent possible for Descriptive Top Level Specifications (DTLS), Formal Top Level Specifications (FTLS), and verification methodologies.

e. Require designs to be full, compilable and executable Ada.

f. Minimize text in documentation.

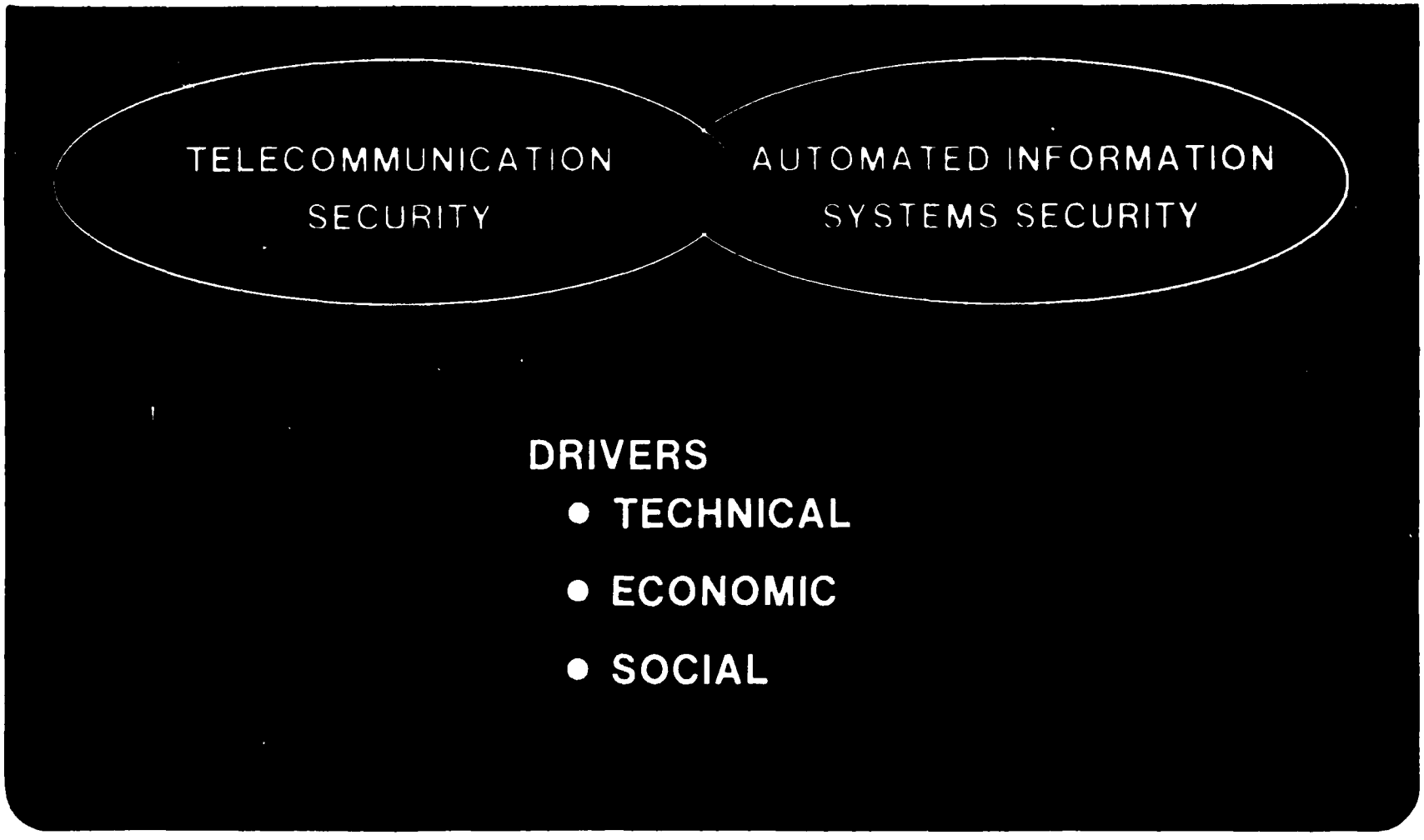
g. Validate Ada compilers for all machines used.

Experience has shown that Ada provides significant savings in lines of code and cost.

The Ada Security Task Force has been merged with the IDA effort. This Workshop is being used by the DODCSC as a forum to formulate issues and track resources which can be used to resolve those issues.

The slides for Col. Greene's presentation follow this page.

CONVERGENCE



THE PRESIDENT'S DIRECTIVE
(NSDD 145, SEP 84)

- RECOGNIZES COMSEC AND AISS CONVERGENCE
- ACKNOWLEDGES
  - INHERENT VULNERABILITIES
  - THREAT TO NATIONAL SECURITY
- RESPONDS TO DEPENDANCE OF ENTIRE SOCIETY ON A VULNERABLE TECHNOLOGY
- DIRECTS CORRECTIVE ACTIVITIES
- ESTABLISHES SINGLE, ACCOUNTABLE LEADERSHIP

CHALLENGE

MUST:

- COMPLETELY CHANGE THE WAY INDUSTRY DESIGNS AND BUILDS
  - WORD PROCESSORS
  - PC'S
  - MINI-COMPUTERS
  - MAINFRAME COMPUTERS
  - DATA BASE MANAGEMENT SYSTEMS
  - LOCAL AREA NETWORKS
  - NETWORK COMPONENTS
  - MULTIMEDIA
- CREATE A NEW TECHNOLOGY BASE
- DISTRIBUTE THAT NEW TECHNOLOGY

AUTOMATED INFORMATION SYSTEM GOALS

RIGHT INFORMATION . . .

AT THE RIGHT PLACE . . .

AT THE RIGHT TIME . . .

GENERAL REQUIREMENTS

- SECURITY
  - CONTROL READING
  - CONTROL ACCESS
  - GUARANTEE SERVICE
- INTEROPERABILITY

DEFENSE GUIDANCE

- 15-YEAR GOAL:
  - INTEROPERABILITY
- 5-YEAR MID-TERM OBJECTIVE:
  - SECURE AIS
  - DEPLOYMENT OF TCB SYSTEMS

MLS CRITERIA

REQUIREMENTS	TCB - 1983	INB - 1985	TDB 1988
POLICY R1 - SECURITY POLICY	REDUCED TO PRACTICE	PRELIMINARY DRAFT	**ASSERTION** OB PROVIDES A NECESSARY AND SUFFICIENT STATEMENT OF FUNDAMENTAL SECURITY REQUIREMENTS
R2 - MARKING			
ACCOUNTABILITY R3 - ID			
R4 - AUDIT		SOME DEVELOPMENT	
ASSURANCE R5 - ENFORCEMENT (OF R1 - R4)			
R6 CONTINUOUS PROTECTION			

MLS CRITERIA

REQUIREMENTS	DAC		
	C1	C2	B1
POLICY			
R1 - SECURITY POLICY			2 INFORMAL MAC POLICY
R2 - MARKING	1 NAMED SUB TO OBJ		1 SENSITIVITY LABELS SUB & OBJ
ACCOUNTABILITY			
R3 - ID	2 ID AUTH SUB	1 ID AT SINGLE-USER GRANULARITY	
R4 - AUDIT		2 AUDIT SEC CRITICAL EVENTS	
ASSURANCE			
R5 - ENFORCEMENT (OF R1 - R4)	3 DAC OF SUB TO OBJ		3 "NOMINAL" MAC
R6 CONTINUOUS PROTECTION	4 TCB PROTECTED DOMAIN 5 DOC	3 MEMORY REUSE	4 TESTING -FUNCTIONAL -PENETRATION

MLS CRITERIA

REQUIREMENTS	MAC		
	B2	B3	A1
POLICY R1 - SECURITY POLICY	1 DTLS	1 FORMAL SECURITY MODEL	1 FTLS
R2 - MARKING ACCOUNTABILITY R3 - ID R4 - AUDIT			
ASSURANCE R5 - ENFORCEMENT (OF R1 - R4)	2 "REAL" MAC	2 "PURE" MAC	2 FORMAL VERIFICATION
R6 CONTINUOUS PROTECTION	5 COVERT CHANNEL	3 TRUSTED FACILITIES	3 STRONG CM CONTROL
	3 TRUSTED PATH -TCB EXEC DOMAIN -DISTRIBUTION CONTROL	4 TRUSTED DELIVERY	
	4 CONFIGURATION CONTROL	5 DOC	4 ADDED DOC

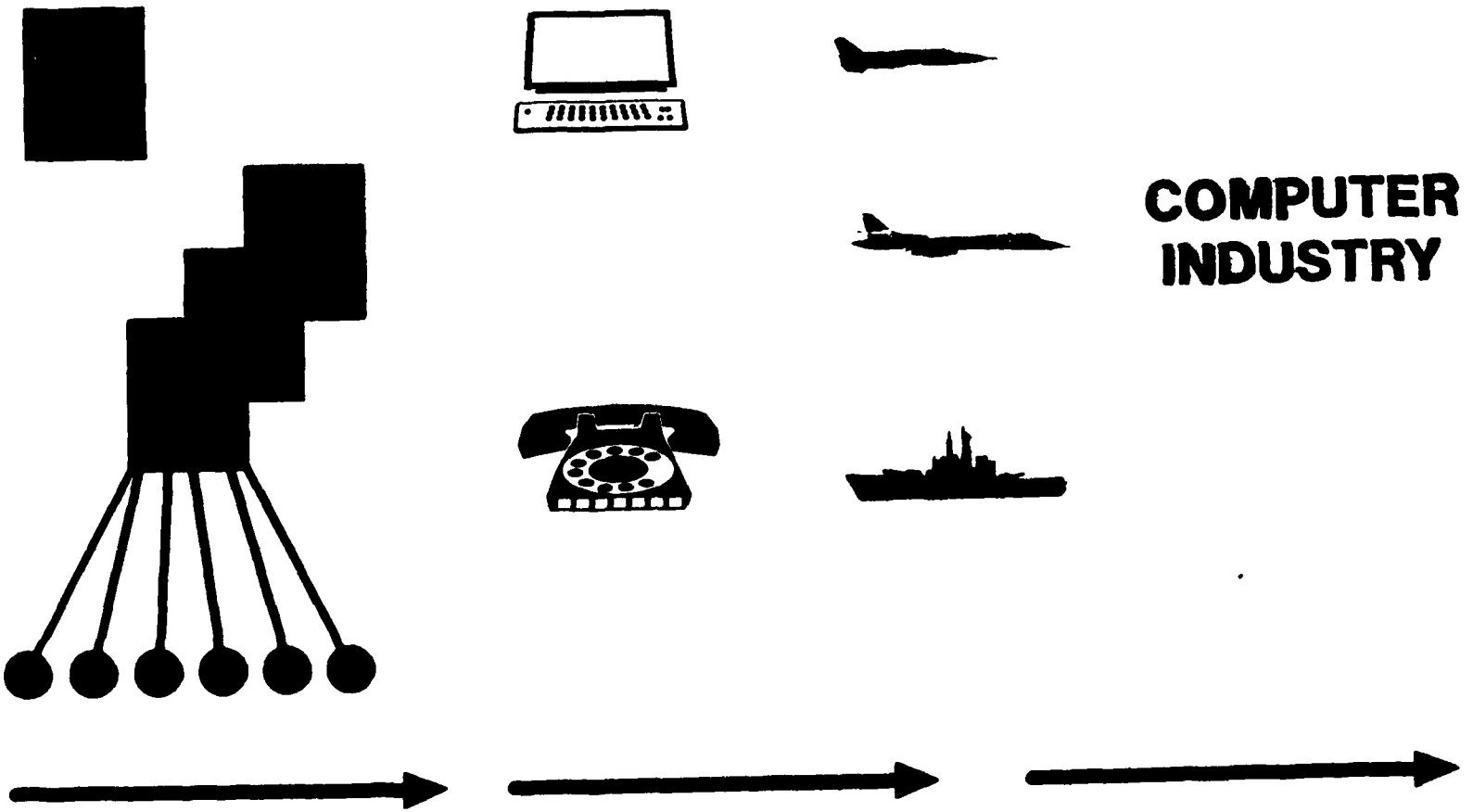
Ada POLICY

- USE & SUPPORT STANDARDS
- MONITOR & INCORPORATE EMERGING STANDARDS
- ALL CODE IN MACHINE - INDEPENDENT Ada
  - USE Ada SYNTAX & SEMANTICS TO MAX EXTENT POSSIBLE
    - SUPPORTING DESCRIPTIONS (DTLS, FTLS, VERIF)
    - COMMAND LANGUAGE (INTERFACE TO MAN-MACHINE INTERFACE LIKE QUERY)
    - DML, DBML
    - SUPPORTING DOCUMENTATION
  - DESIGN INFORMATION AS FULL, COMPILABLE, EXECUTABLE Ada RATHER THAN TEXTUAL

Ada POLICY (CONT'D)

- ALL CODE IN MACHINE - INDEPENDENT Ada (CONT'D)
  - MINIMIZE TEXTUAL DOCUMENTATION
    - INTERNAL
    - NO COMMENTS DUPLICATE EXECUTABLES
    - USE PROMPTS AND HELPS TO REDUCE EXTERNAL USER DOCUMENTATION
    - ALL ELECTRONIC TRANSFERABLE
    - STUDIES ELECTRONIC TRANSFERABLE
    - MULTI-MEDIA ELECTRONIC TRANSFERABLE (GKS, PHIGS, CLIPT)
  - MAXIMIZE USE OF TOOLS
    - FACILITATE SHARING
- NO MACHINES WITHOUT VALIDATED Ada COMPILER

LEVERAGE



1.3 Verification and the Software Life Cycle
- Ann Marmor-Squires, TRW

Charter of the Working Group on the Role of Verification in the Life Cycle

a. Determine the appropriate role(s) for specification and verification technology in the software development life cycle for Ada mission-critical systems development.

b. Describe the relationships between verification technology and other analysis techniques used in the life cycle.

c. Determine the automated support tools needed for the successful application of the technology in its proposed role(s).

d. Recommend means of incorporating verification technology into the life cycle in an effective manner.

e. Recommend near-term projects to be funded.

f. Coordinate efforts with the other working groups.

Verification should be viewed in a broader sense as one part of a whole complex of methods, languages and tools used in the software life cycle. It is important that verification be merged with other methods to give better confidence in the resulting system.

The following issues are important to determining the role of verification in the life cycle:

a. Definition of specification and verification technology. What exactly do we mean by formal specification and verification? What languages, methods and tools are involved?

b. Relationship of verification to METHODMAN.

c. Is there only one life cycle? What are appropriate standards for the life cycle(s)?

d. How will verification be used in the specific application? What properties does one want to verify about the application? What other analysis techniques will be used in addition to verification?

e. How much will it cost to do formal verification?

f. Generic vs. specific methodology and support tools. Verification will play a different role in the life cycle depending upon whether the technology being used is specific to the application (e.g., formal information flow tools for security) or generic (e.g., a verification condition generator).

g. How do we get started on integrating verification into the life cycle? What funding is available for near-term projects?

Configuration management (CM) is particularly important. Both the verified system and the tools used to verify it evolve. As the system evolves, it may need to be re-verified. The evolution of the verification tools must be managed so that new verification technology can be incorporated without making re-verification more difficult (e.g., by incorporating a new verification paradigm which is inapplicable to the system into the tools).

The question of the use of Ada in the development of the Strategic Defense Initiative (SDI) was discussed. Some members of the audience felt that the group present at the Workshop should think about the implications of using Ada in SDI, while others felt that the group already has more than enough to think about. No official decision was made on the matter.

1.4 Near Term Solutions to Ada Verification
- Karl Nyberg, Verdix Corp.

The focus of this Working Group is on adapting existing languages, tools and methods for formal specification and verification to verifying Ada. Examples of existing technology include SPECIAL/HDM and Gypsy. There is, however, no fielded software which has been formally verified. AI systems have been developed using existing technology, but have not been used extensively.

One problem with developing near-term Ada verification systems is that the language may change in 1988. It is not clear what the extent of this change will be, so any Ada verification system developed before then may become obsolete due to language changes. The question is, should we start from scratch in 1988 or build Ada verification systems now and try to adapt in 1988?

Several points were raised in answer to this question. First, by building verification systems now, we can discover some of the verification problems connected with Ada. This will also provide experience with verifying Ada. Even if these early near-term systems are thrown away after 1988, the experience gained will be valuable for building future tools. Second, attempts to build and use systems will help to uncover some of the "fuzziness" of ANSI/MIL-STD-1815A Ada Language Reference Manual, (LRM) which will serve as input to the language change in 1988. Near-term attempts to define a formal semantics for Ada will also help to uncover "fuzzy" Ada features.

At this point, the question was raised whether a formal semantics for Ada must be formulated before Ada verification systems can be built. The general consensus of opinion was that a formal semantics for at least a part of Ada was necessary, but a formal semantics covering all of Ada was not. A formal semantics expressed in terms of axioms and proof rules could be constructed to cover a restricted subset of Ada. These axioms and rules could then be used to build a verification system.

1.4.1 The IBM Clean Room Project
- Tom Kraly, IBM

IBM has experimented with applying semi-formal methods manually (i.e., with no automated tool support) to improve correctness of software. This project is called the "clean room," and is based on the work of Dr. Harlan Mills. The traditional approach to software correctness is to design and implement the software and then to find the bugs and fix them. The IBM clean room project is an attempt to use semi-formal methods during software development so as not to introduce errors

in the first place. The project uses a semi-formal specification language based on set theory. Informal rules of argument are used to reason about software. Software is modeled as state machines.

The clean room project originated in IBM's Federal Systems Division, but is now used throughout IBM. It has been used with Program Design Language (PDL) Ada.

1.5 Ada Specification Languages
- Friedrich von Henke, SRI

Charter of the Working Group on Specification Languages for Ada

The purpose of the Working Group is to discuss Ada-oriented specification languages, with the goal of formulating requirements for such languages and making recommendations for further activities in this area.

Specific topics to be addressed include:

a. The role of specifications and specification languages in the process of producing Ada programs

b. The requirements on a specification language (as opposed to the programming language or design languages)

c. The state of the art of specifying Ada programs

d. Identification of areas of the Ada language for which specification techniques are lacking or insufficient

e. Alternative approaches to the design of specification languages for Ada

As a result of the discussion, the Working Group will formulate requirements for Ada-oriented specification languages and make recommendations for further research and language design efforts.

The activities of the Working Group are to be coordinated with related Working Groups, in particular those addressing the issues of formal semantics of Ada and the role of verification in the software life cycle.

It is impossible to do formal verification without a formal specification language in which to state what you are proving. Therefore, to build an Ada verification system we must have an Ada specification language which is adequate to state the kinds of properties we want to prove about Ada programs.

Specifications can be divided into several areas:

a. Functional: The run-time behavior of the program

b. Structural: Static relationships of various modules in a program

c. Performance: "Hard" real-time properties

d. Security/Safety properties

An area which must eventually be addressed in an Ada specification language is how to specify concurrency and real-time properties. We have little experience in the area of specifying real-time software. In addition, Ada was designed for embedded systems so the specification language should also be able to describe properties of the hardware. This is also an area in which we have little experience.

The current state-of-the-art Ada specification language is ANNA. ANNA currently lacks facilities to specify properties related to concurrency. ANNA is a conservative extension of Ada in that it attempts to use Ada syntax and philosophy as much as possible. Is this the right approach? One can imagine three possibilities:

a. The ANNA approach - make the specification language look as much like Ada as possible, and don't depart from Ada in any significant way.

b. Design a completely different language without attempting to follow Ada.

c. Middle ground - use Ada syntax and philosophy as guidelines but not as dogma.

Although ANNA currently falls into the first category, it could be modified to fall into the third category. The danger in doing this would be that one would have to modify the semantics while keeping the same syntax. It would be better to modify both.

An argument in favor of staying as close to Ada as possible is that this avoids possible incompatibilities between Ada and its specification language. An argument in favor of not being bound by Ada is that it may very well turn out that the properties one wants to prove about a system are not easily stateable in Ada.

A slightly modified form of ANNA is being used in European work on Ada. It would be desirable to have a single standard Ada specification language (e.g., a standard version of ANNA). This specification would help support reuseability of verified software since tools which process the standard specification language could be used on code developed elsewhere.

The term "specification language" is somewhat "fuzzy." It's not clear how a specification language differs from a design language (i.e., Ada PDL). It is especially important to make this distinction clear in Ada. Some people believe that Ada is a specification language. Ada, or an Ada PDL, may be regarded as a

design language, but it is not formal enough to be a specification language. Specification languages must have a high degree of formality to support proofs. In addition, specification languages are supposed to say what the program does rather than how it is done. Using Ada or an Ada PDL might force the specifier to overspecify the program, and would also make it difficult to specify at a high level of abstraction. It would be best if the design language and the specification language were the same language.

1.6 Simplifying Ada Semantics by Restricting Implementers' Options
- Norman Cohen, Softech

Defining a semantics for Ada is difficult because the LRM leaves many things unspecified (e.g., parameter-passing mechanisms, when exceptions are raised, what the effects of certain pragmas are). Norm Cohen presented a proposal for a partial solution to this problem. His proposal introduced the notion of a conservative implementation of Ada. A conservative implementation would be an implementation of full Ada, but with many of the ambiguities of the LRM resolved in a straightforward way. Another way to say this is that a conservative implementation is an implementation which uses a more predictable compiler.

It was suggested that the restrictions that define a conservative implementation might become part of the language definition in 1988 or 1993.

The slides for Mr. Cohen's presentation follow this page.

SIMPLIFYING ADA* SEMANTICS BY RESTRICTING IMPLEMENTERS' OPTIONS

Norman H. Cohen

SofTech, Inc.
705 Masons Mill Business Park
1800 Byberry Road
Huntingdon Valley, PA 19006

(215) 947-8880

NCOHEN@ECLB

*-Ada is a trademark of the U.S. Government, Ada Joint Program Office

EXAMPLES OF UNDERSPECIFIED ADA PROGRAM BEHAVIOR

Given the base type and the values of A and B, will A * B overflow? If so, will Numeric_Error be raised? If so, will it be raised at the point where the expression A * B occurs, or someplace earlier?

Given the declaration "Number_Of_Processors : constant := 5;", will the test

```
if Number_Of_Processors=1 and Arrival_Rate/Service_Rate > 1.0 then ...
```

raise an exception when Service_Rate is zero?

When a procedure call propagates an exception, do the in out and out composite parameters retain their original values?

Can a use clause raise Storage_Error?

Can the declaration "A : String (0 .. 32767);" raise Numeric_Error if overflow occurs during internal computation of the length 32767 - 0 + 1?

ACCEPTABLE FORMS OF NONDETERMINISM

Nondeterminism in multitasking programs

- Verify selective waits in the same way as guarded commands.
- Won't be able to prove timing- or priority-related properties.
- A delay statement can be viewed as a null statement (provided that the delay expression has no side effects)

Order of subexpression evaluation?

When Device_Error and Storage_Error are raised.

CONSEQUENCES FOR FORMAL VERIFICATION

Complicated proof rules.

"Wild cards" in preconditions.

Exponential explosion in the length of formulas.

SOME FAULTY SOLUTIONS

Confine the programmer to a subset of Ada

- Doesn't really solve the problem, which is the amount of leeway given to a compiler in implementing fundamental Ada constructs.
- Certain Ada features (e.g. address clauses) may have to be considered unverifiable anyway.

Build an implementation-dependent verifier

- Proof rules can precisely describe a construct's effect.
- Not cost-effective. (Only a small number of Ada users will benefit.)
- Verifier won't prove portability.
- Erroneous programs and programs with incorrect order dependencies can be verified.
- Proof rules depend on private, undocumented aspects of a compiler's implementation, and could be invalidated by release 2.0 of the compiler.

ANOTHER SOLUTION:
A STANDARD SET OF IMPLEMENTATION RESTRICTIONS

Additional constraints on an implementation, beyond what the Ada Reference Manual requires

An implementation obeying these additional constraints will be called conservative.

General theme of restrictions: Do what the obvious interpretation of the program text suggests, even if this rules out certain optimizations.

A conservative implementation implements neither a subset nor a superset of Ada.

- A conservative implementation must obey all the restrictions in the Ada Reference Manual, plus additional restrictions.
- All conservative implementations are valid Ada implementations, but not vice versa.

EFFICIENCY

Departures from natural semantics were originally allowed to facilitate optimization.

- Efficiency was a major design goal of Ada.

Optimizations should still be allowed for conservative implementations, but only in the presence of pragmas explicitly requesting them.

- A compiler may disregard the presence of the pragma, but not its absence.
- Optimization pragmas are a syntactic flag to the verifier that a portion of the program is not verifiable.
- Use of these pragmas can be restricted to hot spots, leaving most of the program verifiable.
- Analogous to the REORDER option of PL/I.

THE CONSERVATIVE APPROACH TO EFFICIENCY: EXAMPLES AND COUNTEREXAMPLES

EXAMPLES:

- The Suppress pragma.
- The Inline pragma. (Results in a departure from natural compilation-order rules.)
- Unchecked programming.

COUNTEREXAMPLES:

- The Shared pragma. The pragma must be explicitly specified to guarantee natural semantics.
- The option to perform code motion that changes the logical effect of a program.
- The option not to raise Numeric_Error upon overflow.

STANDARDS

Agree on an official standard set of restrictions for conservative implementations

Add a supplement to the ACVC

- All implementations passing the ACVC are conforming implementations.
- All tests passing the supplement as well are conservative implmentations.
- But can appropriate tests be written?

ADVANTAGES OF CONSERVATIVE IMPLEMENTATIONS

Proof rules for conservative implementations will be

- simpler to write and understand
- less expensive to apply during verification

Verifiers based on these proof rules will be valid for all conservative implemenations

Programmer portability: Behavior of an Ada program under one conservative implementation will be predictable to a programmer familiar with another conservative implementation

Readability: Possible departures from expected semantics are explicitly pointed out by optimization pragmas.

CONSEQUENCES

It isn't hard for an implementation to be certified as conservative.

Some time in the future, should the standard definition of Ada be revised to include the additional restrictions on implementations?

- All correct programs would remain correct.
- Compilers performing clever optimizations would have to turn them off in the absence of optimization pragmas.

2 WEDNESDAY MORNING SESSION

2.1 A Proposal for Ada Formal Semantics
- David Luckham, Stanford

There are several reasons why one wants to have a formal semantics for a programming language. First, it provides a standard definition of the language and how the constructs behave for both users and implementors of the language. Second, it provides a basis for reasoning about programs.

There are several approaches to presenting a formal semantics for a programming language that have been used in the past:

a. An interpreter for the language in the language. This is what is done in LISP (page 72 of the LISP 1.5 Manual).

b. An operational definition in terms of abstract machines. An example is the semantics of PL/I, which was defined in terms of an abstract tree automaton in the mid 60's.

c. A denotational definition in terms of Scott domains and recursion equations. This definition was tried for Ada. This denotational semantics did not include tasking.

d. An axiomatic definition in terms of a collection of axioms and a set of proof rules for reasoning about programs. This definition has been done for Pascal.

These approaches have various shortcomings. Formal semantics are generally not "debugged" in the sense that they don't correctly define the behavior of some constructs in some situations. Formal semantics generally do not cover all of the features of the language (e.g., concurrency and real arithmetic). Formal semantics are usually uninformative in that they are hard to read and it is difficult to determine from the formal semantics how a given program will behave.

Dr. Luckham's proposal for a formal semantics for Ada is that there should be two different presentations of the Ada semantics. The first presentation would be a standard instrumented compiler. This would be a compiler which, in addition to compiling programs, would also explain what it is doing in response to users' questions. The second presentation would be an axiomatic proof system which could be used to prove programs with respect to specifications in some standard specification language. Consistency of the two forms of semantics would eventually need to be demonstrated. Conceptually, the proof rules should be derivatives of the semantics of the compiler; in practice, the two would probably be developed in parallel.

It is within the state of the art to build the front end of a standard instrumented compiler. The code generator would require more work, particularly in the area of tasking. The standard implemented compiler would not have to be an efficient compiler; its primary purpose is to provide an executable, informative presentation of the semantics of Ada.

To do the axiomatic proof system, we need to get more experience with specifying Ada programs and with proving properties of concurrent programs. On the basis of this experience, a preliminary standard specification language could be defined, and a proof system could be built. The axiomatic proof system would include specifications of a standard environment, (e.g., a standard I/O package). A test of the proof system would be to see if it could derive the expected behavior of the programs in the Ada compiler validation test suite.

Dr. Luckham's presentation generated a lively discussion with a number of questions. Some concern was expressed that using an instrumented compiler to define the semantics of Ada would be overspecifying the language. One might wish to allow other compilers which are instrumented differently than the standard compiler but are nonetheless regarded as Ada compilers. For example, the Ada/Ed compiler was done in SETL, with the arbitrary implementation choices documented.

Concern was also expressed about the impact on verification of underspecifying the semantics of Ada. This is of particular concern in the area of secure systems. Any indeterminacy in the semantics of Ada should be sufficiently controlled so that meaningful proofs of security properties are possible.

There was some doubt about being able to demonstrate consistency between the two proposed semantics. Consistency could be a problem if the semantics were developed independently. However, if the semantics were developed in parallel, consistency could be maintained through mapping.

Finally, there was some concern that the semantics might become so mathematical that only an expert would be able to use them. The semantics should be written so that the general user can get sensible answers from sensible questions. Whether or not an answer is sensible should be determined in your head or by your peers.

2.2 European Work on Ada Formal Semantics
- Kurt Hansen, Dansk Datamatik Center (DDC)

The DDC developed a formal definition (FD) of 1980 Ada in 1981-82 using the Vienna Development Method (VDM). This definition was not as mathematically formal as it could have been--- there is no formal definition of VDM itself. Nonetheless, a validated compiler was derived from this FD.

Another activity of DDC was the RAISE project. This was a project to develop Ada support tools, such as interpeters and verification tools.

Previous work on formal definitions of Ada has used denotational semantical style. These definitions are not very readable, partly due to the fact that a denotational semantics always specifies a complete model, which essentially forces you to overspecify). One of the goals of current DDC work is to produce a more readable style for an FD. Ultimately, DDC would like to be able to derive a natural language explanation of the FD directly from the FD. It is not intended that most people who want to use Ada will read the FD. Most people will learn about Ada from books written by people who have read the FD.

Another goal is to provide an unambiguous definition of Ada. The LRM has many ambiguities which must be resolved in the process of creating an FD. The approach that DDC has taken is that where there is an obvious way to resolve an ambiguity, it is incorporated into the FD. When there is no obvious resolution, some resolution is chosen and an explanation of the ambiguity is included in the FD. The FD has also been cross-referenced to the LRM.

The technical description of the FD is divided into static semantics and dynamic semantics. The static semantics deals with the relationships between program units, whereas the dynamic semantics deals with execution behavior. A static semantics of Ada is well-defined. A dynamic semantics includes sequential execution, parallel execution (concurrency) and I/O, and is much less well-defined. The process has been to start from the LRM text, add static semantics and then add the dynamic semantics. The static semantics consists of denotational-style domain equations plus some abstract data types. The static semantics defines whether a program is well-formed and how overloading is resolved.

The dynamic semantics is formed by adding transformation rules to the static semantics. The dynamic semantics of purely sequential execution (no concurrency) can be read as an ordinary denotational semantics. The part of the dynamic semantics dealing with concurrency is expressed in the SMoLCS (Structured Monitored Linear Concurrent Systems) methodology. SMoLCS is based on labelled transition systems. It defines the semantics of processes in terms of their behavior rather than their state.

The model of a dynamic environment and storage has been done, but there is currently no certainty that it works in all cases.

The FD project is currently working on formally defining a subset of Ada to test the expressive power of the tools. This is intended to evolve to a full ANSI/MIL-STD-1815A Ada specification by the end of the calendar year 1986. After the full FD is formulated, the next step is to make a correlation between the FD and the LRM. This correlation will be important for making the FD readable and understandable. After the correlation is made, the next aim of the project will be to create an informal explication of the FD (e.g., a textbook).

Other aims of the project include:

a. Building tools to support a machine-readable LRM

b. Creating educational courses and texts

c. Maintaining liaison with standards groups (e.g. ISO WA9, Language Maintenance Committee, ANSI)

d. Comparing the Ada FD and the ACVC (are they consistent?)

e. Mapping the FD into a SETL program for testing

SETL might have been adopted as the language in which to express the FD, but it needs to become more flexible.

Ada is very strongly supported in Europe. The Commission of the European Communities (CEC) sponsors Ada work through several projects, including the European Strategic Programme for Research and Development of Information Technologies (ESPRIT) and the Ada Multi-Annual Programme (Ada MAP). Research targets for 1985-86 include the relationship of Ada to knowledge bases. One area which has not been strongly addressed by these projects is proof systems for verification. A project to prove some properties of Ada/Ed was considered at one time but was abandoned as too expensive.

The slides for Mr. Hansen's presentation follow this page. Additional material can be found in Appendix B.

THE DRAFT FORMAL DEFINITION OF ANSI/MIL–STD 1815A:

ADA*

*Ada ia a registered trademark of the U.S. Government (AJPO)

History

Aims of the Formal Definition

Technical Description

Project Description

Summary

Dansk Datamatik Center Member Companies: (1 Jan 85)

I/S Datacentralen

Kommunedata

Danish Defence Research Est.

Jutland Telecom

A/S Regnecentralen

SDS Datacenter

Telecommunication Research Lab

O. K. Data

Commission of the European Communities (CEC)

E uropean

S trategic

P rogramme for

R esearch and development in

I nformation

T echnologies

Multi

A nnual

P rogramme

CHILL

- Student project at Technical University of Denmark
- Masters thesis on static semantics
- Final version accepted as supplement to the CCITT Z.200 recommendation (October 1981)
- DDC has derived a CHILL compiler from the formal definition

Ada

- Several masters theses on Ada Semantics (80)
- Formal Definition of 1980 Ada (81–82) using VDM
- CEC multiannual programme (DDC, Olivetti, CR)
- DDC derived their validated Ada compiler

Lead DDC to:

— "Formal methods to Industry" project

(ESPRIT project: RAISE)

— Formal definition of ANSI/MIL–STD 1815A Ada

(MAP project)

RAISE

- 115 man years
- 5 year project
- DDC, STCplc, NBB, ICL
- Development of scientific foundation
- Method Development
- Specification Language and Tools
- Technology Transfer to industry
- Industrial Trials

RAISE TOOLS

- Interpreters / rapid prototypes
- Development processors (transformation)
- Proof and Verification (of development step)
- Proof of properties of specifications
- Document processor

(also base tools as syntax directed EDITORS)

Aims of the Formal Definition

- Highly Readable (STYLE)
- Implementation independent
- Unambiguous definition

Target Groups

- Language Designers and Reviewers
- STD People
- Implementors
- Validators
- Educators
- Programmers
- Scientist

Ada Formal Definition Requirements

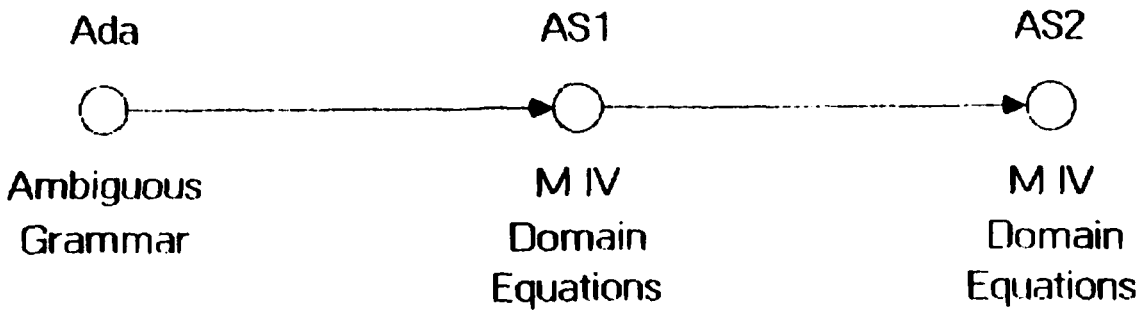
- Legal Contract
- Consistent and Complete
- Comprehensive and Precise
- Correct and Believed Correct
- Accessible and Referenceable
- Permissive
- Implementation Bias
- Processor Development
- Validation
- Proof Systems
- Mechanisable
- Rapidly Prototypable
- Correlateable
- Document Derivable

Technical Description

1) Static Semantics

2) Dynamic Semantics

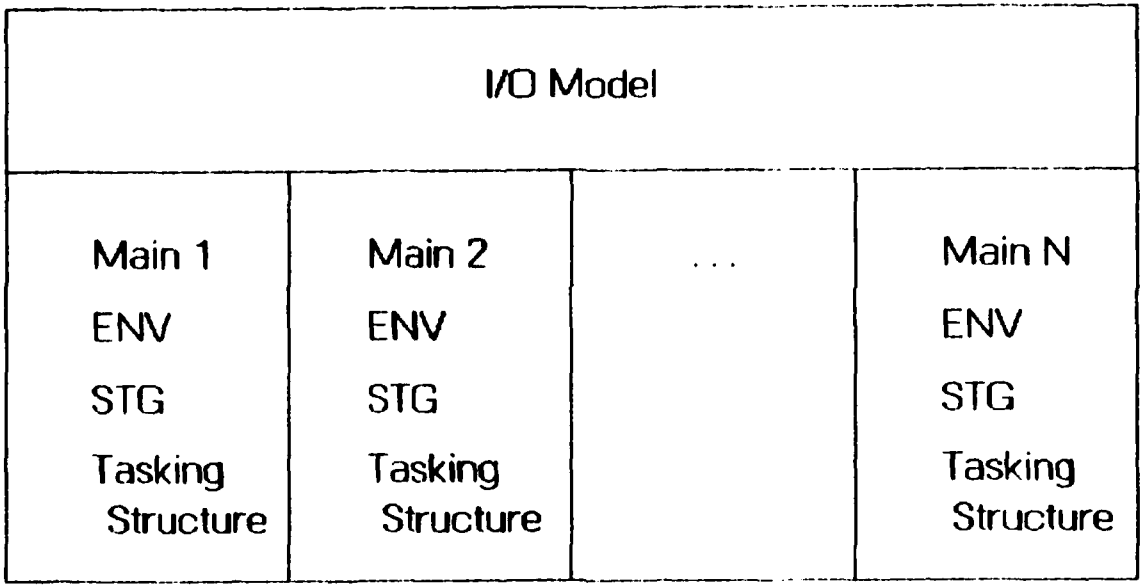
   - Sequential
   - Parallel
   - Input/Output



LRM Text

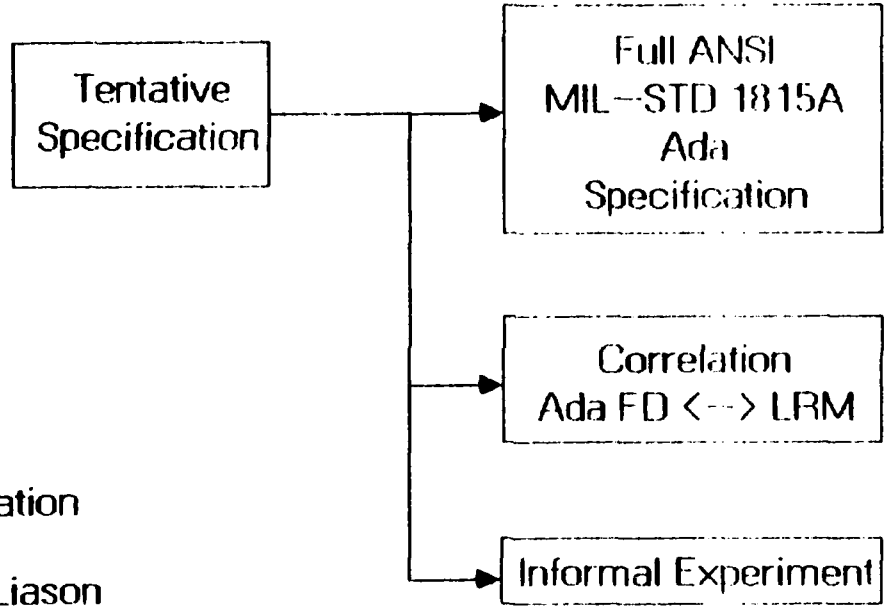
Static Semantics

Dynamic Semantics


I/O Model
Main 1
ENV
STG
Tasking Structure
Main 2
ENV
STG
Tasking Structure
. . .
Main N
ENV
STG
Tasking Structure

Project Description

- 13 man years
- 2 year project (85--86)
- DDC, CRAI
  IEI/CNR
  Technical University of Denmark
  University of Pisa
  University of Genoa



- Tools
- Education
- STD Liason
- ACVC vs. Ada FD (study)
- Mapping to N.Y.U. (study)

3 WEDNESDAY AFTERNOON SESSION

3.1 A Notation for Ada Proof Rules
- Norman Cohen, Softech

Traditional notations for proof rules (e.g., the notations of Hoare or Dijkstra) have certain drawbacks that complicate formal verification. Norm Cohen presented a notation that is a variation on Dijkstra's notation and has particular advantages for Ada proof rules. The slides for Mr. Cohen's presentation follow this page.

A NOTATION FOR ADA* PROOF RULES

Norman H. Cohen

SofTech, Inc.
705 Masons Mill Business Park
1800 Byberry Road
Huntingdon Valley, PA 19006

(215) 947-8880

NCOHEN@ECLB

*-Ada is a trademark of the U.S. Government, Ada Joint Program Office

HOARE'S NOTATION FOR PROOF RULES

$$\frac{\begin{array}{l} P \text{ and } In\_Choice\_List\ (v, L_1) \text{ implies } Q_1 \\ \quad \ldots \\ P \text{ and } In\_Choice\_List\ (v, L_n) \text{ implies } Q_n \\ \\ \{Q_1\}\ SS_1\ \{R\} \\ \quad \ldots \\ \{Q_n\}\ SS_n\ \{R\} \end{array}}{\begin{array}{l} \{P\} \\ \\ \text{case } v \text{ is} \\ \quad \text{when } L_1 \Rightarrow SS_1 \\ \quad \ldots \\ \quad \text{when } L_n \Rightarrow SS_n \\ \text{end case} \\ \\ \{R\} \end{array}}$$

PROBLEMS WITH HOARE'S NOTATION

- Can be used to justify a proof, but not to direct automatic construction of one.
- It is hard to ascertain whether all situations have been accounted for.
- Awkward to handle expressions with side-effects.
- Awkward to handle nonlinear program flow (exceptions, loop exits, returns).

DIJKSTRA'S NOTATION FOR PROOF RULES

wp (<u>case</u> v <u>is</u> <u>when</u> L_1 => SS_1 ... <u>when</u> L_n => SS_n <u>end</u> <u>case</u>, P) =

(In_Choice_List (v, L_1) and wp (SS_1, P)) or

... or

(In_Choice_List (v, L_n) and wp (SS_n, P)

- Constructive.
- Easy to validate that <u>wp</u> is defined for all possible arguments.
- Does not address side effects or nonlinear program flow.

ENHANCEMENTS TO DIJKSTRA'S NOTATION

precondition (construct, postcondition, status)

The construct may be a statement, expression, declaration, etc.

The postcondition may be any predicate.

The status is one of the following:

null: normal completion

(an exception name):
completion with that exception raised

exit N: completion with an exit for loop N or a return from subprogram N in progress

- Only possible if the construct is a statement, statement sequence, etc.
- Applies to both compound and simple statements.

EXAMPLE: INDEXED COMPONENTS OF ONE-DIMENSIONAL AR

wp (A(v), P, null)

= [weakest precondition if A is evaluated first] and
  [weakest precondition if v is evaluated first]

= wp (A, wp (v, P and v in A'Range, null), null) and

  wp (v, wp (A, P and v in A'Range, null), null)

For any exception e other than Constraint_Error,

wp (A(v), P, e)

= [weakest precondition if A is evaluated first] and
  [weakest precondition if v is evaluated first]

= ([weakest precondition if A raises e] or
   [weakest precondition if A is normal and v raises e]) and

  ([weakest precondition if v raises e] or
   [weakest precondition if v is normal and A raises e])

= (wp (A, P, e) or wp (A, wp (v, P, e), null)) and
  (wp (v, P, e) or wp (v, wp (A, P, e), null))

EXAMPLE CONTINUED

```
wp (A(v), P, Constraint_Error)

   = [weakest precondition if A is evaluated first] and
     [weakest precondition if v is evaluated first]

   = ([weakest precondition if A raises Constraint_Error] or
      [weakest precondition if A is normal and v raises Constraint_Error] or
      [weakest precondition if A and v are normal and the range check
       raises Constraint_Error]) and

     ([weakest precondition if v raises Constraint_Error] or
      [weakest precondition if v is normal and A raises Constraint_Error] or
      [weakest precondition if v and A are normal and the range check
       raises Constraint_Error)

   = ( wp (A, P, Constraint_Error) or
       wp (A, wp (v, P, Constraint_Error), null) or
       wp (A, wp (v, P and v not in A'Range, Constraint_Error), null) ) and

     ( wp (v, P, Constraint_Error) or
       wp (v, wp (A, P, Constraint_Error), null) or
       wp (A, wp (v, P, Constraint_Error), null) )
```

PREDICATE TRANSFORMATION FUNCTIONALS

F: constructs x statuses -> (predicates -> predicates)

is defined by: F [c, s] (P) = ~~wp (c, s, P)~~ wp(c, P, s)

Example:

F [A(v), null] (P) =
F [A, null] (F [v, null] (P and v in A'Range)) and
F [v, null] (F [A, null] (P and v in A'Range))

Composition, denoted by &, may then be defined as follows:

(F [c_1, s_1] & F [c_2, s_2]) (P)
= F [c_1, s_1] (F [c_2, s_2] (P))
= wp (c_1, wp (c_2, P, s_2), s_1)

Example:

F [A(v), null] (P) =
(F [A, null] & F [v, null]) (P and v in A'Range) and
(F [v, null] & F [A, null]) (P and v in A'Range)

COMPOSITIONS CAN BE READ LEFT TO RIGHT TO DESCRIBE ORDER OF PROCESSING

Suppose that processing of a construct C consists of processing subconstructs C_1, C_2, and C_3 in that order.

If P is to hold afterward, with normal completion,

wp (c_3, P, null) must hold just before c_3 is processed

wp (c_2, ●, null) must hold just before c_2 is processed

wp (c_1, ●, null) must hold just before c_1 is processed

wp (c_1, wp (c_2, wp (c_3, P, null), null), null)

= F [c_1, null] (F [c_2, null] (F [c_3, null] (P)))

= (F [c_1, null] & F [c_2, null] & F [c_3, null]) (P)

USE OF FUNCTIONAL EQUATIONS TO DEFINE SEMANTICS

Instead of writing

$$F\,[c, \text{null}]\,(P) =$$

$$(F\,[c_1, \text{null}] \;\&\; F\,[c_2, \text{null}] \;\&\; F\,[c_3, \text{null}])\,(P)$$

simply write

$$F\,[c, \text{null}] = F\,[c_1, \text{null}] \;\&\; F\,[c_2, \text{null}] \;\&\; F\,[c_3, \text{null}]$$

<u>Paraphrase</u>: The effect of normal completion of construct c is equivalent to normal completion of subconstruct c_1, followed by normal completion of subconstruct c_2, followed by normal completion of subconstruct c_3.

OTHER MACHINERY TO FACILITATE FUNCTIONAL EQUATIONS

Null_Transformer (P) = P for all predicates P

Constant_Transformer [P] (Q) = P for all predicates P and Q

For all predicates P and all predicate transformers F_1 and F_2:

$(F_1$ and $F_2)$ (P) = F_1 (P) and F_2 (P)
$(F_1$ or $F_2)$ (P) = F_1 (P) or F_2 (P)
$(F_1$ xor $F_2)$ (P) = F_1 (P) xor F_2 (P)
(not F_1) (P) = not (F_1 (P))

(F_1 implies F_2) (P) = not F_1 (P) or F_2 (P)

Add_Conjunct [P] = Constant_Transformer [P] and Null_Transformer

(<u>QUIZ</u>: What is the value of Add_Conjunct [P] (Q)?)

INDEXED COMPONENTS REVISITED

F [A(v), null] =

F [A, null] & F [v, null] & Add_Conjunct [v in A'Range] and
F [v, null] & F [A, null] & Add_Conjunct [v in A'Range]

For an exception e other than Constraint_Error,

F [A(v), e] =

(F [A, e] or F [A, null] & F [v, e]) and
(F [v, e] or F [v, null] & F [A, e])

F [A(v), Constraint_Error] =

(F [A, Constraint_Error] or
F [A, null] & F [v, Constraint_Error] or
F [A, null] & F [v, null] & Add_Conjunct [v not in A'Range]) and

(F [v, Constraint_Error] or
F [v, null] & F [A, Constraint_Error] or
F [v, null] & F [A, null] & Add_Conjunct [v not in A'Range])

3.2 Secure Systems Working Group
- Margie Zuk, MITRE

Purpose: To study the impact of Ada on the design and implementation of secure systems.

Up to now, language issues have not had a big impact on secure system design. Ada, however, has many features that previous languages have not. As has been discussed earlier in the conference, there are many uncertainties about the additional features and how they will affect security design. These features fall into three categories:

a. Language constructs - what are the security issues connected with Ada constructs such as tasks and exceptions?

b. Run-time Support Library (RSL) - the Ada Run-time Support Library is like a small operating system itself. How should the run-time support library for a secure system in Ada be designed?

c. Compiler Issues - how can we be sure that the unpredictabilities in the definition of Ada do not undermine the security of the system?

All of these questions need to be addressed before Ada can be used with confidence in secure designs.

Although there are complications introduced by using Ada for secure systems, there are also benefits. The Ada features to support software engineering (e.g., packaging, separate compilation units) make it more probable that Ada code will be correct. Other languages have no support for software engineering. In addition, the fact that Ada is a high level language with features like strong typing makes it superior to unstructured, untyped languages like assembly language.

The security community is interested in "zero-term" solutions, i.e., what can we do with the technology that is available today? Ms. Zuk's suggestion was to restrict the use of Ada constructs in order to enhance the understandability and verifiability of programs (e.g., the "conservative" compiler approach presented by Norm Cohen). The slides for Ms. Zuk's presentation are follow this page.

SECURE SYSTEMS WORKING GROUP

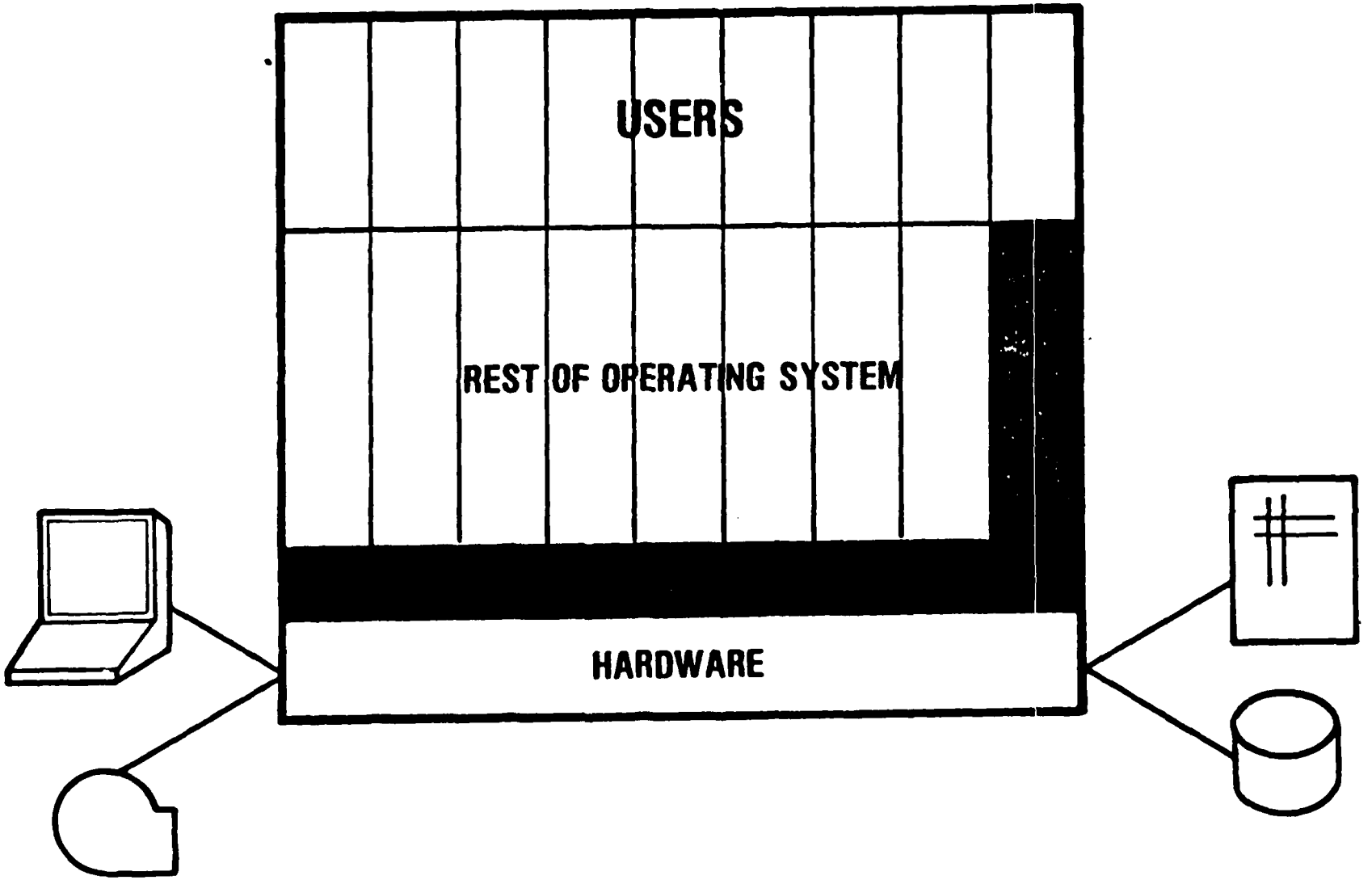
PURPOSE

TO DETERMINE THE IMPACT OF ADA ON THE DESIGN AND IMPLEMENTATION OF SECURE SYSTEMS

OVERVIEW

- SECURE OPERATING SYSTEM OVERVIEW
- AREAS OF CONCERN
- GOALS OF THE SECURE SYSTEMS WORKING GROUP

TRUSTED COMPUTING BASE (TCB)



DOD Trusted Computer System Evaluation Criteria

CSC-STD-001-83, 15 August 83

"Orange Book"

Includes: Criteria
Rationale & Guidelines

Evaluation Criteria Divisions

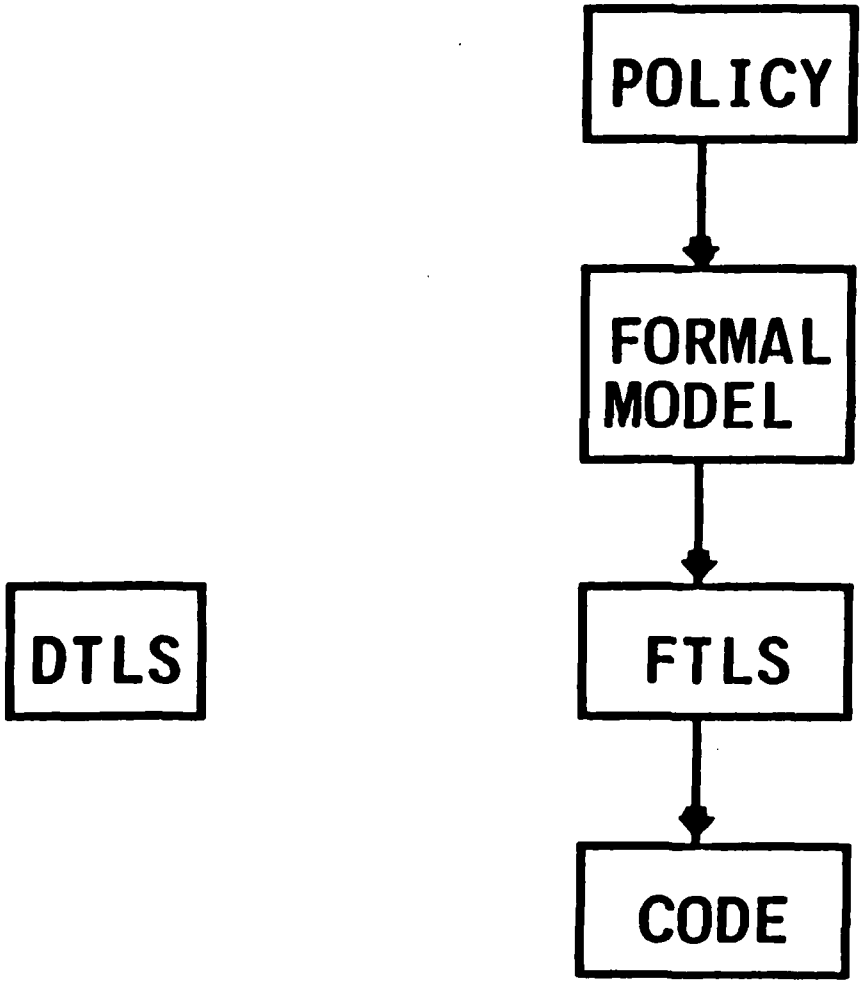
A — Verified Protection

B — Mandatory Protection

C — Discretionary Protection

D — Minimal Protection

THE VERIFICATION PROCESS



GOALS OF SECURE SYSTEMS WORKING GROUP

- DELINEATE THE FEATURES OF ADA THAT INTRODUCE NEW SECURITY CONCERNS
  - o LANGUAGE FEATURES
  - o RUNTIME SUPPORT LIBRARY
  - o COMPILER ISSUES
- STUDY "CONSERVATIVE" COMPILER APPROACH FOR SECURITY
- DETERMINE RESTRICTIONS ON LANGUAGE USE FOR SECURE SYSTEM IMPLEMENTATIONS

GOALS OF SECURE SYSTEMS WORKING GROUP (CONT.)

- STUDY BENEFITS OF USING RESTRICTED SET OF ADA FEATURES OVER OTHER LANGUAGES
- STUDY RSL IMPACT ON SECURE SYSTEM DESIGN AND VERIFICATION
- IDENTIFY AND TRACK ONGOING EFFORTS IN SECURE ADA SYSTEMS

AREAS OF CONCERN

- RUNTIME SUPPORT LIBRARY
- COMPILER IMPLEMENTATIONS
- COMPLEXITY OF THE LANGUAGE ITSELF
  - -ADA TASKING MECHANISM

As part of the secure systems talk, representatives from Softech and Verdix gave talks on specific Ada RSL's.

3.2.1 The Softech RSL
- Juern Jurgens, Softech

The Softech RSL runs on top of UNIX BSD 4.1 and its sole responsibility is to handle signals. In an architecture like the Nebula architecture (MIL-STD-1862B Nebula Instruction Set Architecture, 03 January 1983), the Softech RSL allows some Ada features to be implemented directly in hardware (e.g., task switching is supported directly by the Nebula hardware). However, the Collection pragma is not implemented in the component of the RSL that does storage management. The slides for Mr. Jurgens' presentation follow this page.

SOFTECH

RUNTIME SUPPORT LIBRARY

MOST IMPORTANT INTERFACES OF THE RSL

EXECUTING ADA PROGRAM
(GENERATED CODE)

RSL

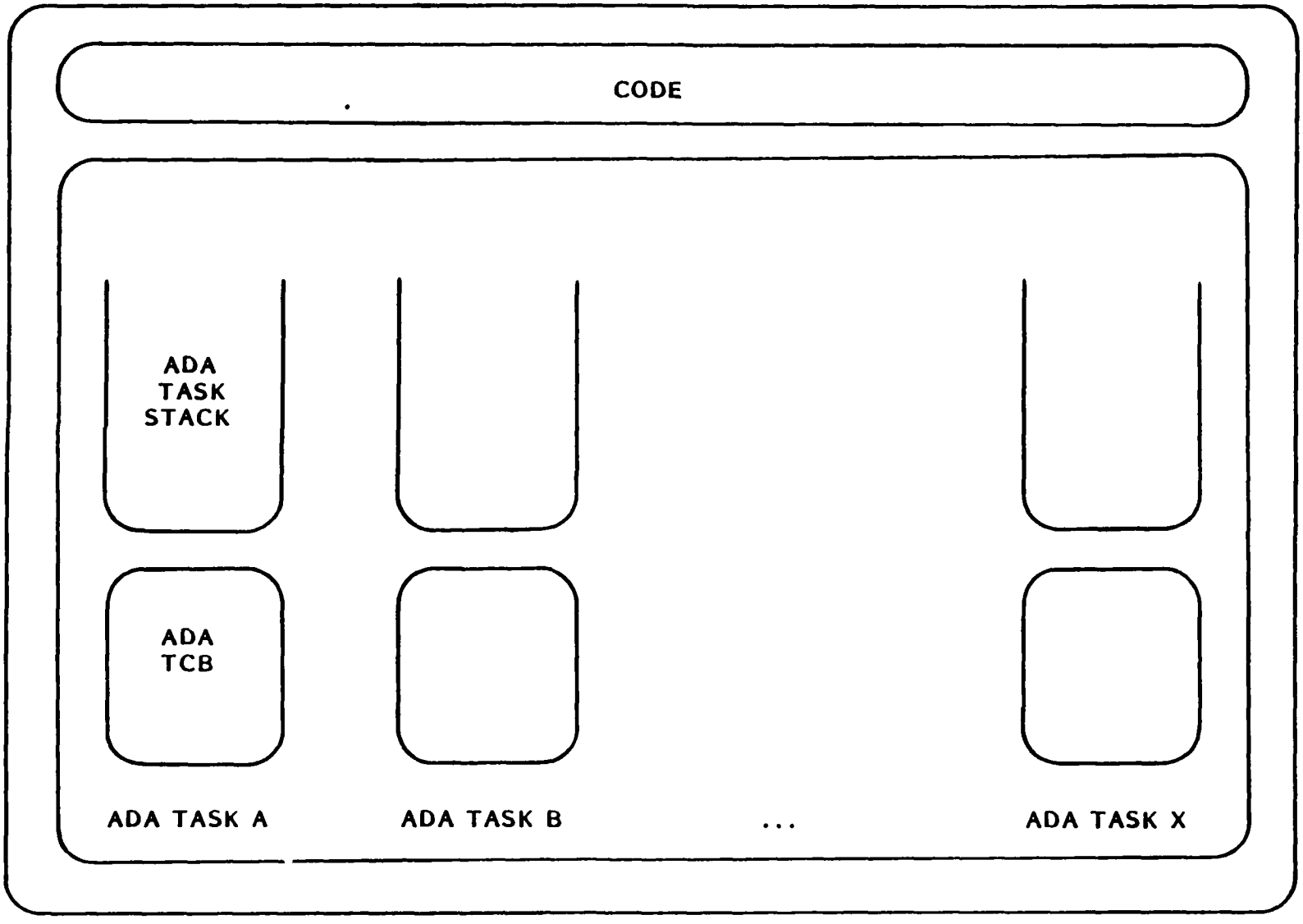
UNIX BSD 4.1

DIPS MACHINE

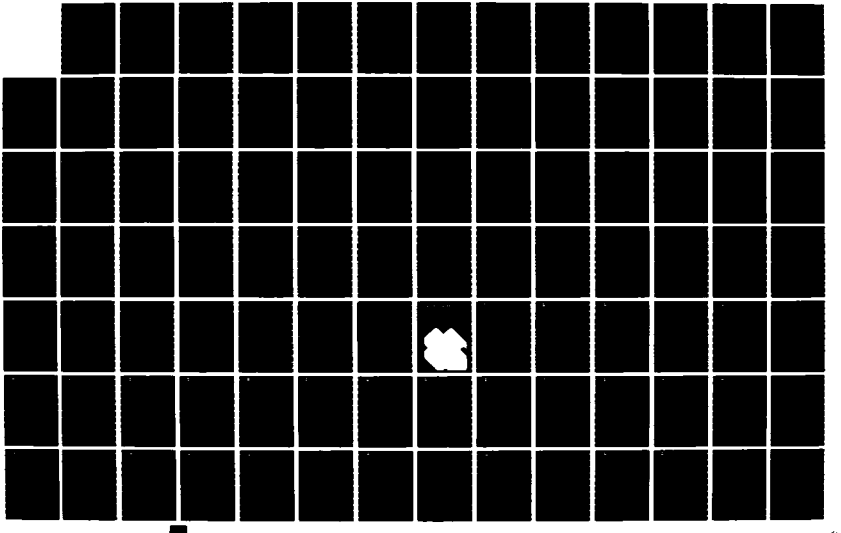
COMPONENTS OF THE RSL

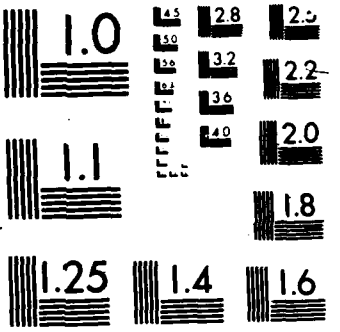
- TASK DISPATCHER : BLOCKING/WAKING OF ADA TASKS
- INTERRUPT DELIVERY : HANDLING OF UNIX SIGNALS
- DYNAMIC STORAGE ADMINISTRATION : ALLOCATION/DEALLOCATION OF STORAGE
- TASK MANAGEMENT : ADA RENDEZVOUS, TASK ACTIVATION, TASK TERMINATION
- I/O REQUEST HANDLING : DIRECT_IO, SEQUENTIAL_IO, TEXT_IO
- MISCELLANEOUS SUPPORT : GENERATED CODE SUPPORT
- EXCEPTION DELIVERY : FINDING OF EXCEPTION HANDLERS, TRANSFER OF CONTROL
- SYSTEM ACTIVATION : START EXECUTION OF THE ADA PROGRAM

ADA TASK IMPLEMENTATION

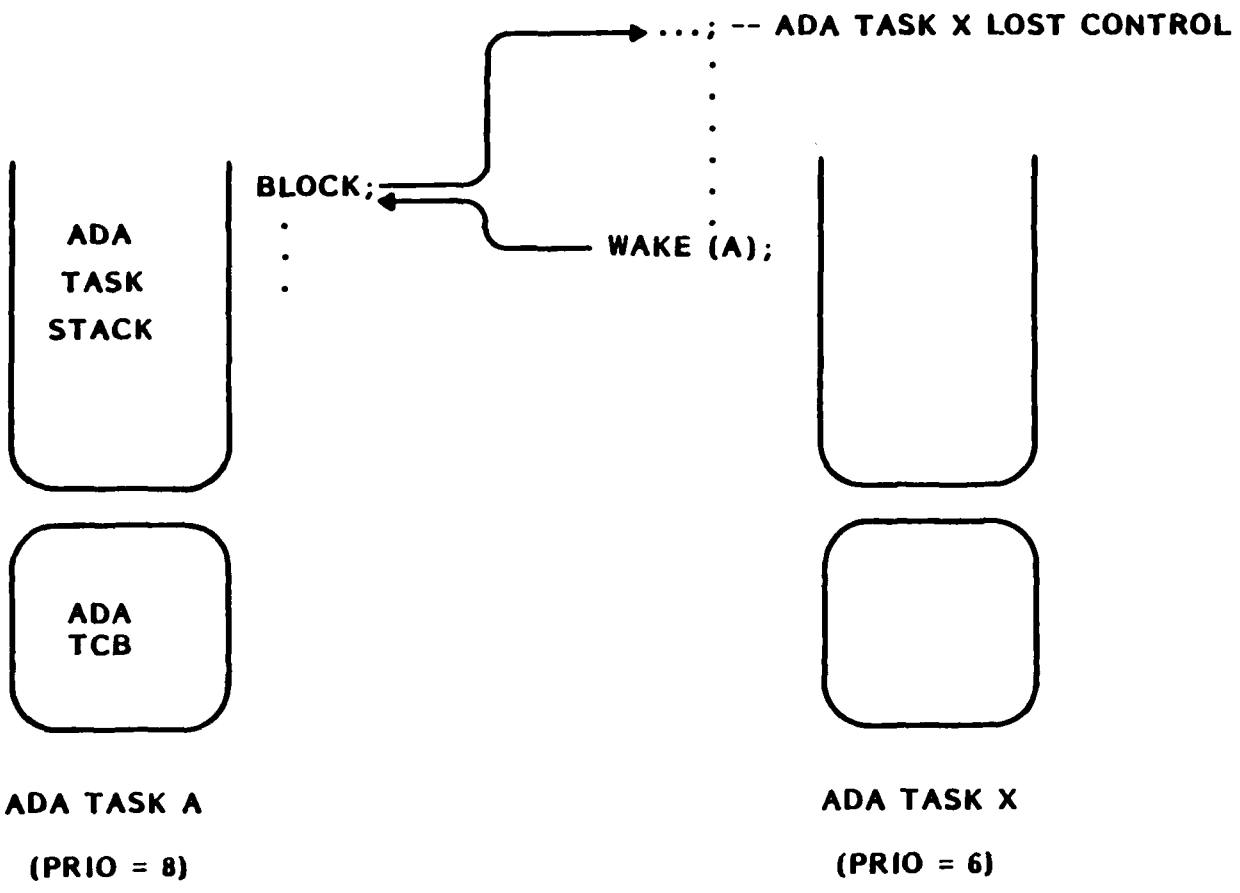


AD-A172 747 PROCEEDINGS OF THE IDA (INSTITUTE FOR DEFENSE ANALYSES) 2/4
WORKSHOP ON FORMA..(U) INSTITUTE FOR DEFENSE ANALYSES
ALEXANDRIA VA W T MAYFIELD ET AL. NOV 85 IDA-M-135
UNCLASSIFIED DECL IDAH030579 IDA/HQ-85-30579 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0
4.5
5.0
5.6

TASK SWITCH


...; -- ADA TASK X LOST CONTROL
BLOCK;
WAKE (A);
ADA
TASK
STACK
ADA
TCB
ADA TASK A
(PRIO = 8)
ADA TASK X
(PRIO = 6)

TASK DISPATCHER

- STRICTLY PRIORITY DRIVEN
- READY LIST : ONE SUBQUEUE PER PRIORITY LEVEL
- SYSTEM TASKS HAVE THEIR OWN READY LIST,
  ARE ABOVE ALL USER TASKS
- FREEZE / THAW : DISABLE / ENABLE SUBQUEUES IN READY LIST

INTERRUPT DELIVERY

```
SIGSYS (SIGALRM, AL_HDLR); -- ESTABLISH HANDLER FOR SIGALRM

.
.
.
DELAY (1.0);

      ALARM (2);

      BLOCK;

        .
        .
        .

      BLOCK;

ADA TASK A

(PRIO = 8)
```

```
WAKE (A);
.
.
.
SIGALRM

void AL_HDLR ( )
 { SAVE CONTEXT OF TASK X;

   UPDATE DELAY LIST;

   ...

   RESUME TASK A;
 }

ADA TASK X

(PRIO = 6)
```

HANDLING OF UNIX SIGNALS

SIGALRM ⟶ ADA DELAY IMPLEMENTATION, TIMER SERVICES

SIGFPE ⟶ ADA EXCEPTION NUMERIC_ERROR

SIGILL ETC. ⟶ ADA EXCEPTION PROGRAM_ERROR

ASYNCHRONOUS OPERATION COMPLETION ROUTINES

TIMER COMPLETION ROUTINES

} INVOKED IF REGISTERED WITH THE RSL

DYNAMIC STORAGE ADMINISTRATION

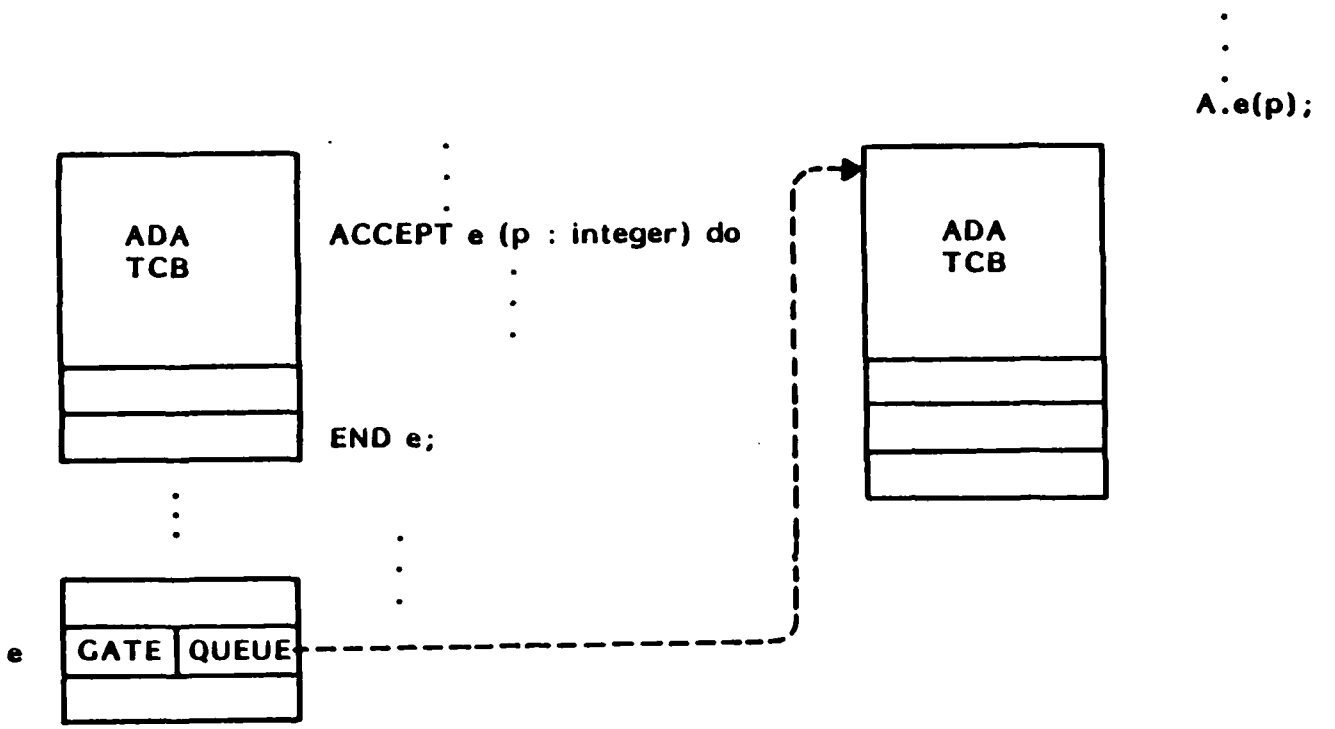
ALLOCATE/DEALLOCATE STORAGE THROUGH MALLOC / FREE

USED FOR:
- RSL INTERNAL USE (ADA TCBs, ADA TASK STACKS...)
- DYNAMICALLY ALLOCATED OBJECTS (NEW)

HEAP INFORMATION BLOCK : HEADER OF LINKED LIST OF OBJECTS, BELONGING TO A GIVEN "BLOCK"

COLLECT : DEALLOCATE ALL OBJECTS BELONGING TO THE CURRENT BLOCK

ADA RENDEZVOUS IMPLEMENTATION



SELECT STATEMENTS

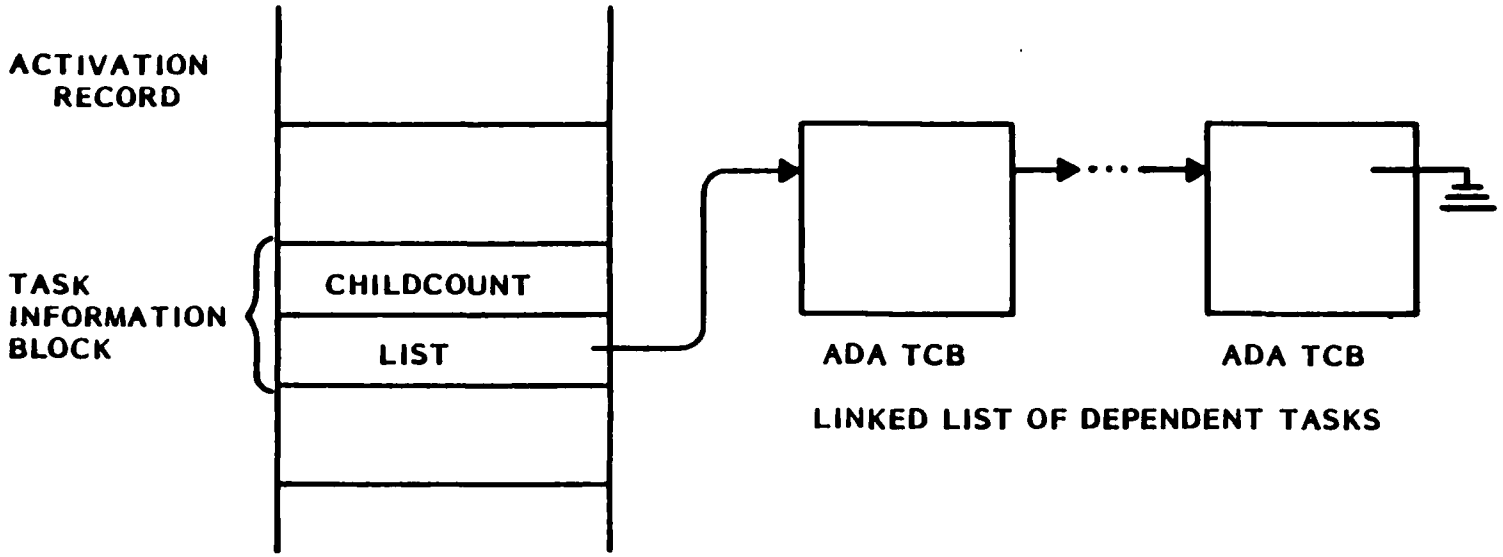
- TIMED ENTRY CALL : ENTER ENTRY QUEUE AND DELAY QUEUE
- CONDITIONAL ENTRY CALL : DEGENERATE TIMED ENTRY CALL
- SELECTIVE WAIT :
  - GENERATED CODE EVALUATES WHEN CONDITIONS
  - BUILD SELECT TABLE .
  - OPEN ALL RELEVANT ENTRY GATES

TASK CREATION AND ACTIVATION

- CREATE ACTIVATION CHAIN
- GENERATED CODE MAKES TASK DESCRIPTOR, TASK OBJECT RECORD
- CREATE_TASK ALLOCATES ADA TCB, TASK STACK, LINKS TASK INTO ACTIVATION CHAIN
- ACTIVATE_TASKS (ACTIVATION CHAIN)
  - INITIALIZES CONTEXT REPOSITORY FOR EACH TASK IN CHAIN,
  - PLACES EACH TASK IN READY LIST,
  - BLOCKS CREATING TASK
- INTRODUCE_TASK
  - CALLED BY EACH NEW TASK,
  - SIGNALS END OF ACTIVATION,
  - LAST CALL UNBLOCKS CREATING TASK

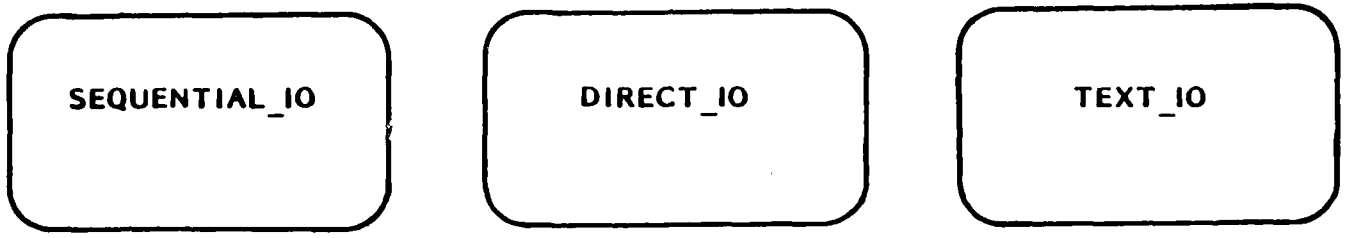
TASK TERMINATION

A UNIT MUST NOT RETURN BEFORE ALL ITS DEPENDENT TASKS ARE TERMINATED




```
WAIT FOR CHILDCOUNT = 0;
RETURN DEBRIS FOR EACH DEPENDENT TASK;
IF THIS UNIT IS A TASK BODY THEN
     DECREMENT PARENT UNIT'S CHILDCOUNT
END IF;
```

I/O REQUEST HANDLING



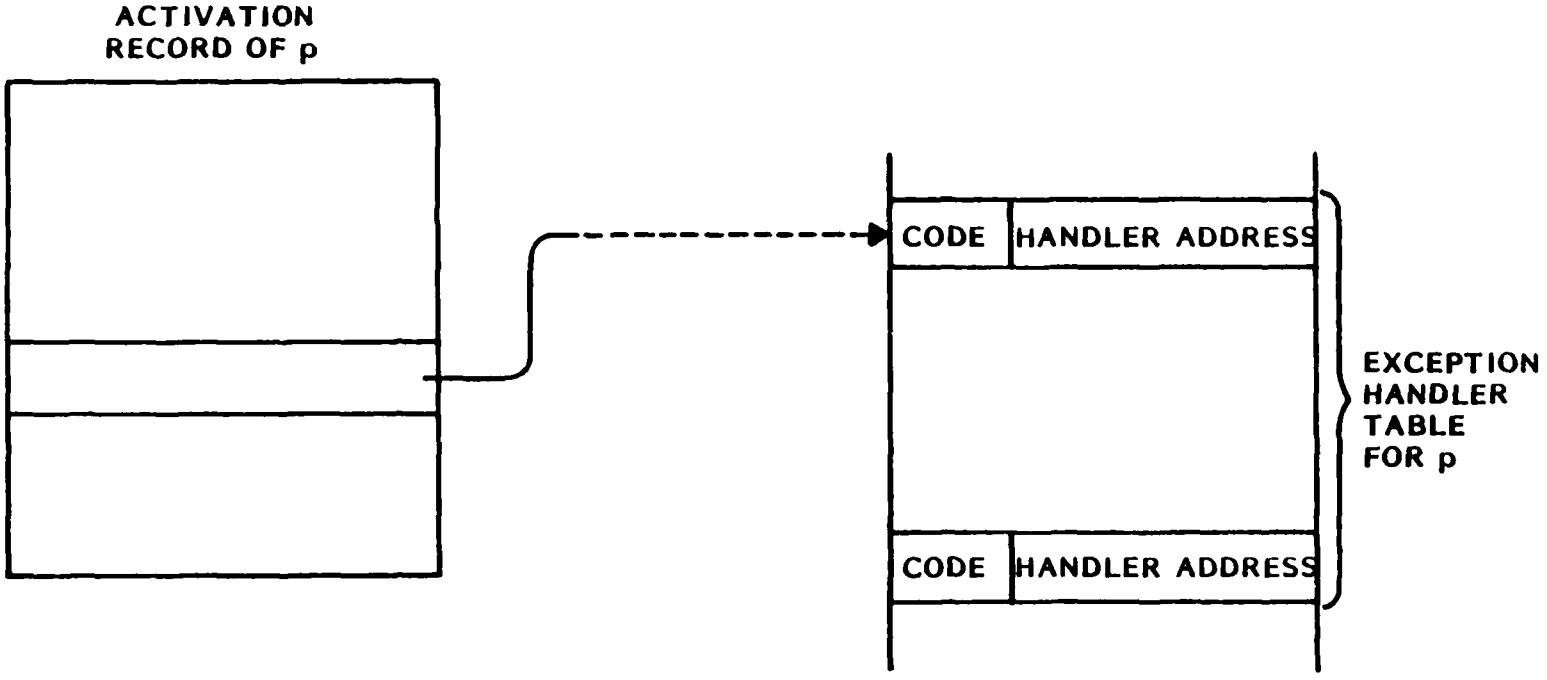
SERVICES PROVIDED BY PRIM_IO

1) CREATING A FILE
2) DELETING A FILE
3) OPENING A FILE
4) CLOSING A FILE
5) READING FROM A FILE
6) WRITING TO A FILE
7) RETRIEVING THE CURRENT FILE INDEX
8) SETTING THE CURRENT FILE INDEX
9) REWINDING A FILE
10) GETTING A FILE'S CURRENT LENGTH
11) GETTING A FILE'S (HOST SYSTEM) NAME

- THESE SERVICES ARE ADA PROGRAM SYNCHRONOUS
- THESE SERVICES ARE AVAILABLE TO THE KAPSE

EXCEPTION DELIVERY

INVOKED BY:	- GENERATED CODE	RAISE
	- INTERRUPT DELIVERY	FORCE_EXCEPTION (NUMERIC_ERROR)
	- TASK ABORTION	FORCE_EXCEPTION (E_TERMINATE)



IF HANDLER IS FOUND, CONTROL IS TRANSFERRED TO IT

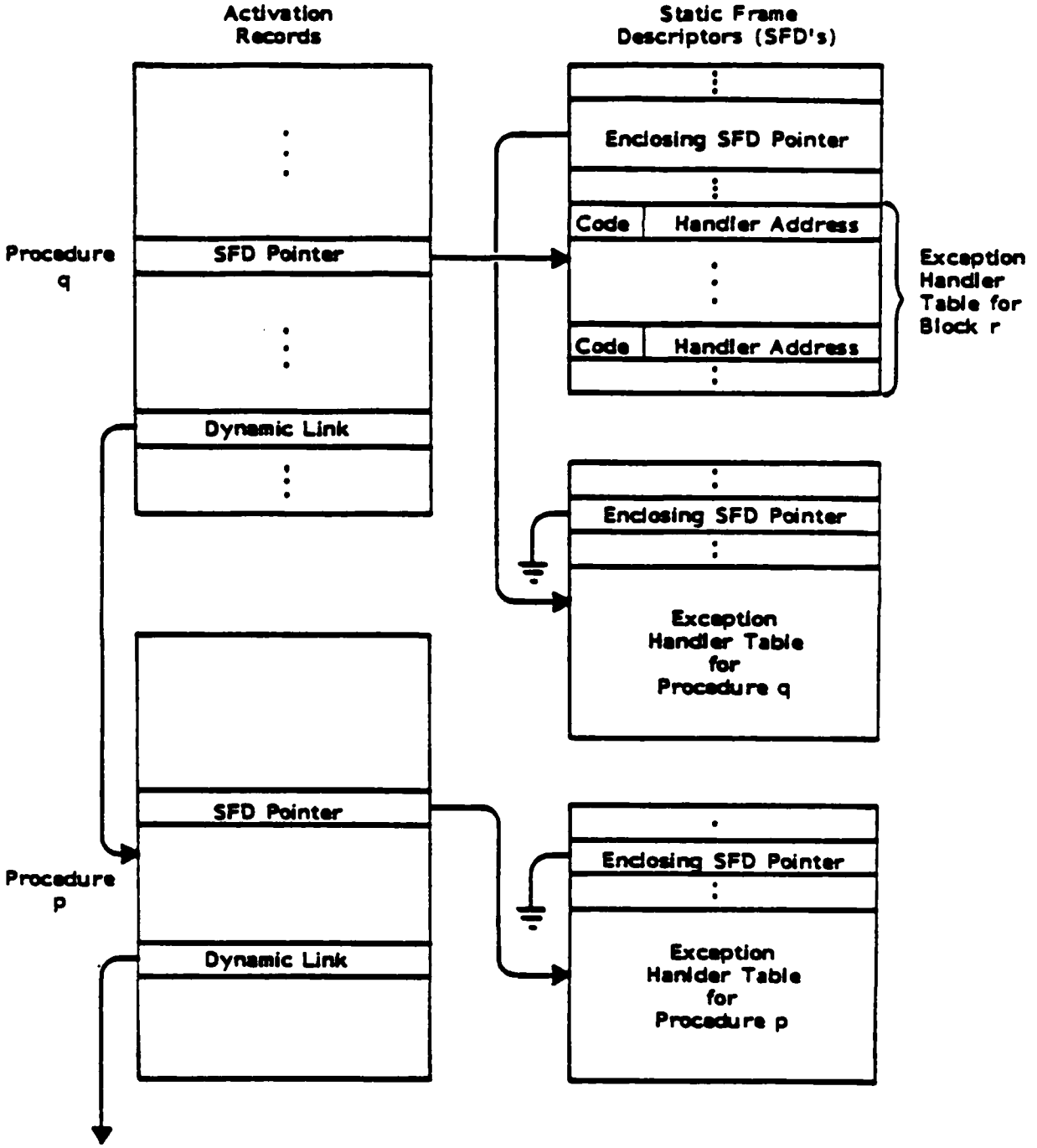
EXCEPTION PROPAGATION

IF NO MATCHING HANDLER IN CURRENT UNIT,

- WAIT FOR TERMINATION OF DEPENDENT TASKS
- FREE DYNAMIC STORAGE
- PROPAGATE EXCEPTION INTO "CALLING TASK" IF APPLICABLE
- RESUME SEARCH FOR HANDLER IN DYNAMIC PARENT

IF NO MATCHING HANDLER ANYWHERE IN CURRENT TASK, TERMINATE TASK

DATA STRUCTURES FOR EXCEPTION PROPAGATION



SYSTEM ACTIVATION

- SET UP INITIAL SIGNAL HANDLER
- SET UP STACK FRAME FOR ANONYMOUS TASK
- ELABORATE LIBRARY UNITS OF RSL
- ELABORATE USER LIBRARY UNITS
- INVOKE ADA MAIN PROGRAM

3.2.2 The Verdix RSL
- Omar Ahmed, Verdix Corp.

The security issues for an Ada RSL include both inter-program and inter-task security. The interaction between two different Ada programs of different security levels running on the same machine is an issue that is external to Ada. The concerns are the same as they would be for two programs written in any languages.

Interactions between two Ada tasks of different security levels that are part of the same Ada program is much more Ada-specific. Inter-task security may require some kind of "level" pragma to indicate the security levels of the various tasks within a single program. Such a pragma would direct the compiler to check for certain kinds of interactions (i.e., rendezvous between a SECRET task and a TOP SECRET task or shared memory between two tasks of different levels). These checks could probably be done at compile time.

Supporting such pragmas would, however, effectively change the language, since such pragmas would forbid certain interactions that would otherwise be legal. A program might compile successfully without the pragmas but not with them. It is more manageable to adopt the convention that a single Ada program runs at a single level, with all tasks within it at the same level.

One approach to limiting interactions between Ada programs at a single level is the Rushby separation kernel approach. In this approach, programs of different levels are isolated from each other in separate domains, and can only communicate through the separation kernel. The separation kernel only allows very strictly controlled imter-program communication. The separation kernel approach could be regarded as a zero-term solution to inter-program security.

The Ada LRM says nothing about inter-program communication. Inter-program communication could be added in the RSL (e.g., a "mailbox" facility). Programs could also share memory. As further functionality (e.g., shared memory, file systems) is added to the RSL, more complicated security mechanisms have to be built into the RSL, and verification becomes more complicated. Verifying the security-relevant portion of an RSL is a near term goal which is boundable.

The slides for Mr. Ahmed's presentation follow this page.

100

Verdix Ada Runtime Systems

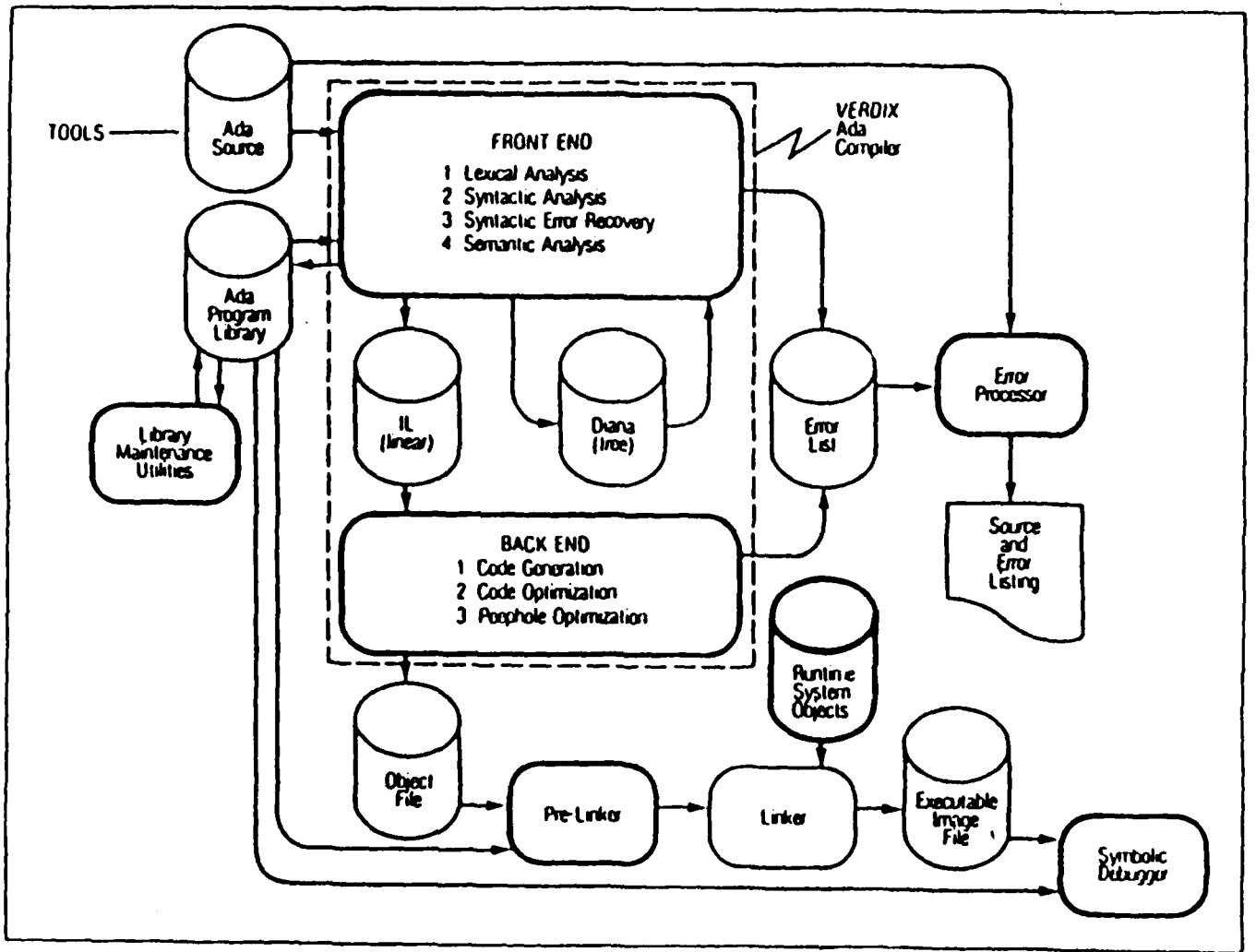
Omar Ahmed
Verdix Corporation

- Brief Overview of the Verdix Ada Development System
- The Verdix Ada Runtime System
- Security Issues for an Ada Runtime System

Verdix

Ada Development System

VERDIX Ada Development System



— VERDIX provided.

VERDIX Ada DEVELOPMENT SYSTEM

- Production-Quality Ada Compiler
- Screen-Oriented Symbolic Debugger
- Configurable Runtime Systems
- Programmer Support Tools
- Rehostable and Retargetable

VERDIX Ada DEVELOPMENT SYSTEM FEATURES

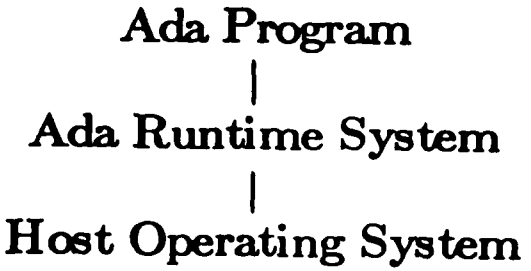
- Production-Quality Compiler
  - — Fast (~1000 lpm on Vax 780 for "real" programs)
  - — Friendly (800+ different errors and warnings)
  - — Accurate (Messages reference LRM by paragraph)
  - — Reliable (Front-end field test termed "excellent")
  - — Complete (e.g., subunits in generics, multi-library)
- Validated 12 December 1984 (ACVC 1.5)

VERDIX Ada DEVELOPMENT SYSTEM FEATURES

- Debugger
  - — Screen-Oriented
  - — Source Level OR System Level
  - — Downline Debugging Versions
  - — Complete Support of Breakpoints, Displays, Value Alteration, etc.
  - — Multi-Language: Ada & C now

Verdix Ada Runtime System

Verdix Ada Runtime System Structure



Functionality

- Memory Management
- Tasking
- Exception Handling
- Input/Output
- Debugging

Memory Management

- Initialization of memory structures (heaps, stacks, pools)
- Allocation
- Deallocation (*NO* garbage collection)

Tasking

- Provides an Ada program with a virtual view of its processor
- Tasking services are layered on the host operating system

  ⇒ Each Ada program is represented as a single host operating system process

Tasking (cont.)

- Creation and Initialization (stacks, TCBs, etc.)
- Activation
- Suspension
- Scheduling (prioritized)
- Accepts
- Selects
- Termination (normal and abnormal)

Exception Handling

- Hardware
- Software
- Storage Errors
- Tasking Errors

Input/Output

- Text I/O
- File I/O

⇒ Supported through the host operating system

Debugging

- Runtime system organized to provide efficient debugging access to Ada tasking information

  ⇒ "Stand-aside" debugging model

Ada Runtimes for Embedded Systems

- Near-term: VRTX (Hunter & Ready)
- Soon: Verdix Ada Real-time Executive
  - — Written in Ada
  - — Sysgen configurable
  - — Support for Motorola 68000 family, Intel 86 family, and MIL-STD-1750A
  - — Multiprocessor support
  - — No intermediate runtime system layers

Security Issues

for an

Ada Runtime System

Functional areas of concern are:

- Inter-Program Security
- Inter-Task Security
- I/O Interfaces, particularly outputs
- Asynchronous events (interrupts)

Specific Ada Runtime System Concerns

- Memory Management
  - — Access to global and local objects
  - — Clearing of information for memory reuse
- Tasking
  - — Scope of called tasks
  - — Capabilities for rendezvous
- Input/Output
  - — Outputs across a security perimeter
- Asynchronous Events
  - — Interruption of processing
  - — Preemption of a transaction
  - — Correctness of the process state

Verification Requirements

- To provide inter-tasking security, the Ada Runtime System needs to be verified
- To provide inter-program security, the host executive needs to be verified
- A general, verified Ada Runtime System seems unlikely since different security policies will generate different:
  - — Security models
  - — FTLS's
  - — Assertions
  - — Implementation correspondence

4 THURSDAY MORNING SESSION

The Thursday morning session consisted of summaries by the Working Group Chairs of the workshop activities relevant to their working groups, and recommendations for actions to be taken in the area of formal verification of Ada.

Richard Platek announced that an attempt was being made to create a SIGAda Committee for Formal Methods, and that 90 minutes had been reserved at the next SIGAda meeting in Minneapolis, Minnesota, for the Working Group Chairs to report on the Workshop. The hope was expressed that this committee would not be isolated from other SIGAda committees.

Much of the Workshop was devoted to the issue of a formal semantics for Ada. The Europeans have done much more in this area than has been done in the United States. Several proposals for work in the area of Ada semantics were put forth, including:

a. Identify and standardize a set of restrictions defining a "conservative" implementation of Ada that would simplify the semantics.

b. Develop multiple formal definitions of Ada aimed at facilitating proofs.

c. Develop a standard instrumented compiler to answer programmers' and implementers' questions.

If several different formal semantics are developed, there should be some way of reconciling them or demonstrating their consistency. Decisions like what form to present the semantics in and whether it should be a semantics for full Ada or only a restricted subset should be made on the basis of attempts to actually create a semantics, rather than on a priori judgement about what is feasible.

Some concern was expressed about whether it was appropriate to propose standards (e.g., a standard instrumented compiler, a standard formal definition) at this time. There was a general consensus that there is a need for a standard formal semantics and a standard mechanism for reasoning about programs; it was felt that these two items were not the same thing, and should be distinguished. There was also a general consensus that pursuing David Luckham's proposal for a standard instrumented compiler would be useful.

Friedrich von Henke presented the following recommendations for work in Ada specification languages:

a. Experiments with specifying programs in ANNA should be carried out and the experience evaluated, with the goal of

eventually arriving at a generally accepted specification language at the code/package level.

b. Languages for specifying concurrency, real time behavior and floating point arithmetic should be explored. Much basic research is needed here.

c. Development of Ada-oriented requirements, design and specification languages should be further explored. ANNA is a language for design and code specification. Design/specification languages for Ada should integrate advanced concepts, and should be based on a formal semantics of Ada.

The point was made that decisions about languages, in particular what constitutes an "Ada-oriented" design/specification language, must be based on experience. It was suggested that if the design/specification language is too divergent from the Ada philosophy, it will be impractical to use.

Margie Zuk presented the following recommendations for work in secure systems in Ada:

a. Delineate the features of Ada that introduce new security concerns (i.e., concerns that are specific to Ada).

b. Investigate the "conservative" compiler approach for security. What impact would optimization pragmas have on assurance that a system is secure?

c. Determine what restrictions should be placed on the use of Ada for secure systems design and implementation. This would include formulating a rationale for any specific restriction.

d. Study the security and verification issues related to the Ada RSL.

e. Identify and track ongoing efforts in secure Ada systems (e.g., the Army Secure Operating System (ASOS)).

John McHugh presented the following recommendations for work in near term Ada verification systems (0-4 years):

a. Develop prototype verification systems built around existing specification languages; experiment with the prototypes by applying them to real problems.

b. Investigate the use of semi-formal methods, e.g., the IBM Clean Room project.

c. Consider Ada-specific verification problems, both in the abstract and from the point of view of existing systems (e.g., what problems would crop up if SCOMP were redone in

Ada). The latter will help to produce a really useable subset of Ada.

d. Consider constraints on RSL's and code generation to enhance confidence in verification.

APPENDIX A

ADA VERIFICATION MAILING INFORMATION

APPENDIX A

Ada Verification Mailing Information

Since verification impacts not only coding activities but all development activities, it is desirable that many groups continue to be informed about the progress of these workshops. Therefore, the account ADA-VERIFY has been created on USC-ECLB and will be used as a central repository for Ada Verification announcements, files, etc. The list shown below has also been established on USC-ECLB to encourage the exchange of ideas:

Ada-VERIFICATION-LIST

Messages that are sent to this list will be received by all of the individual electronic addresses that are included in the Mailing Directory.

The Mailing Directory is provided as the remainder of Appendix A. It is a directory of workshop participants and other interested parties along with their postal, telephonic, and electronic addresses.

NOTE: The AJPO is planning to move all ECLB accounts to ISI. Addresses will be (name) @Ada-20 as of 22 November 1985.

* Those persons who attended the 2nd Workshop are noted in the Mailing Directory with an asterisk.

Mailing Directory

Bernard Abrams
Grumman Aerospace Corporation
Mail Station 001-31T
Bethpage, NY 11714
(516) 575-9487
ABRAMS@USC-ECLB

* Omar Ahmed
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

* Eric R. Anderson
TRW DSG (R2/1134)
One Space Park
Redondo Beach, CA 90278
(213) 535-5776
TRWRB!TRWSPP!ERA@BERKELEY

* Dr. Thomas C. Antognini
MITRE Corporation
Mailstop B330
Burlington Road
Bedford, MA 01730
(617) 271-7294
SECURITY!TCA@MITRE-BEDFORD or
TCVB@MITRE-BEDFORD

Charles Applebaum
1058 Boyurgogne
Bowling Green, OH 43402
(419) 352-0777
CHA@MITRE-BEDFORD

Krzystof Apt
Thomas J. Watson Research Center
P. O. Box 218
88-K01 Route 134
Yorktown Heights, NY 10598
(914) 945-2923

Terry Arnold
Merdan Group
P.O. Box 17098
San Diego, CA 92117
MERDAN@ISI

Ted Baker
Department of Computer Science
Florida State University
Tallahassee, FL 32306
(904) 644-2296

David Elliot Bell — DBELL@MIT-MULTICS
Trusted Information Systems, Inc.
3060 Washington Road
Glenwood, MD 21738
(301) 854-5889

Dan Berry
3531G Boelter Hall
Computer Science Department
School of Eng. and Appl. Science
Los Angeles, CA 90024
(213) 825-2971

Edward K. Blum — BLUM@ECLB
Mathematics Department
University of Southern California
Los Angelos, CA 90089
(213) 743-2504

* Alton L. Brintzenhoff — SCI-ADA@USC-ISI
SYSCON Corporation
3990 Sherman Street
San Diego, CA 92110
(619) 296-0085

* Dr. Dianne Britton — HELBIG@ISI
RCA Adv. Tech. Labs
ATL Building
Moorestown Corporate Center
Moorestown, NJ 08057
(609) 866-6654 or (609) 924-3253

* Dr. R. Leonard Brown — BROWN@AEROSPACE
M1/112
The Aerospace Corporation
P. O. Box 92957
Los Angeles, CA 90009
(213) 615-4335

Richard Chan — RCHAN@USC-ECL (bad)
Hughes Aircraft Co.
P. O. Box 33
FU-618/P115
Fullerton, CA 92634
(714) 732-7659

* Norman Cohen — NCOHEN@ECLB
SofTech, Inc.
705 Masons Mill Business Park
1800 Byberry Road
Huntingdon Valley, PA 19006
(215) 947-8880

* Paul M. Cohen — PCOHEN@ECLB
Ada Joint Program Office
OUSDRE/R&AT
Pentagon Room 3D139 (Fern Street)
Washington, DC 20301-3081
(202) 694-0211

Richard M. Cohen — COHEN@UTEXAS-20
Institute for Computing Science
2100 Main Bldg.
University of Texas
Austin, Texas 78712
(512) 471-1901

Michael D. Colgate — FREEMAN@FORD-COS1
Ford Aerospace & Comm. Corp.
10440 State Highway 83
Colorado Springs, Colorado 80908

* Mark R. Cornwell — CORNWELL@NRL-CSS
Code 7590
Naval Research Lab
Washington, D.C. 20375
(202) 767-3365

Major Terry Courtwright — COURT@MITRE
WIS/JPMO/ADT
7726 Old Springhouse Road
Washington, DC 20330-6600
(202) 285-5056

* Dan Craigen — CMP.CRAIGEN@UTEXAS-20
c/o I. P. Sharp Associates
265 Carling Avenue
Suite 600
Ottawa, Ontario, Canada K1S 2E1
(613) 236-9942

Steve Crocker, M1-101 — CROCKER@AEROSPACE
The Aerospace Corporation
P.O. Box 92957
Los Angeles, CA 92957
(213) 648-6991

John J. Daly — WCOXTON@USADHQ2
USAISSAA
2461 Eisenhower Avenue
Alexandria, VA 22331-0700

Tom Dee
Boeing Commercial Airplane Co.
P. O. Box 3707
MS 77-21
Seattle, WA 98124
(206) 237-0194

Jeff Facemire — FACEMIRE%TI-EG@CSNET-RELAY
Texas Instruments
P.O. Box 801
M/S 8007
2501 West University
McKinney, TX 75069
(214) 952-2137

* John C. Faust — FAUST@RADC-MULTICS
RADC/COTC
Griffiss AFB, NY 13441
(315) 330-3241

Gerry Fisher
IBM Research 35-162
P. O. Box 218
Yorktown Heights, NY 10598
(914) 945-1677

Roy S. Freedman — FREEDMAN@ECLB
Hazeltine Corporation
Greenlawn, NY 11740
(516) 261-7000

James W. Freeman
Ford Aerospace & Comm. Corp.
Mailstop 15A
10440 State Highway 83
Colorado Springs, CO 80908
(303) 594-1536

Mark Gerhardt — MSG@MITRE-BEDFORD
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7839

Chuck Gerson
Boeing Aerospace Co.
Mailstop 8H-56
P.O. Box 3999
Seattle, WA 98124

Helen Gill
MITRE
Mailstop W459
1820 Dolly Madison Boulevard
McLean, Virginia 22102
(703) 883-7980

Kathleen A. Gilroy
Software Prod. Solutions, Inc.
P. O. Box 361697
Melbourne, FL 32936

Virgil Gligor
Department of Electrical Engineering
University of Maryland
College Park, Maryland 20742
(301) 454-8846

Donald I. Good — GOOD@UTEXAS-20
2100 Main Building
The University of Texas at Austin
Austin, TX 78712
(512) 471-1901

Ronald A. Gove — GOVE@MIT-MULTICS
Booz, Allen & Hamilton
4330 East West Highway
Bethesda, MD 20814
(301) 951-4624

* Inara Gravitis — GRAVITIS@ECLB
SAIC
1710 Goodridge Drive
McLean, VA 22202
(703) 734-4096 or (202) 697-3749

* Col. Joseph S. Greene, Jr. — JGREENE@USC-ISI
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6818

David Gries — GRIES@CORNELL
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
(607) 256-4052

David Guaspari
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020
RPLATEK@ECLB

* J. Daniel Halpern
SYTEK Corp.
1225 Charleston Road
Mountain View, CA 94043
(415) 966-7300
SYTEK@SRI-UNIX or
MENLO70!SYTEK!DAN@BERKELEY

* Kurt W. Hansen
Dansk Datamatik Center
LuudToftevej 1C
DK2800 Lyngby
Denmark
PHONE: ++ 45 2 872622
KHANSEN@ECLB

* Scott Hansohn
Honeywell Secure Comp. Tech. Center
Suite 130
2855 Anthony Lane South
St. Anthony, MN 55418
(612) 379-6434
HANSOHN@HI-MULTICS

* Larry Hatch
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6790
HATCH@TYCHO

Linn Hatch
IBM
17100 Frederick Heights
Gaithersburg, MD 20879

* Brian E. Holland
DoDCSC, C3
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6968
BRIAN@TYCHO

Ray Hookway
Dept. of Computer Eng. & Science
Case Institute of Technology
Case Western Reserve University
Cleveland, OH 44106
(216) 368-2800
HOOKWAY%CASE@CSNET-RELAY

Paul Hubbard
Dept. of Computer Eng. & Science
Case Institute of Technology
Case Western Reserve University
Cleveland, OH 44106
(216) 368-2800
HOOKWAY%CASE@CSNET-RELAY

Jim Huitema
National Security Agency
R831
Ft. Meade, MD 20755
(301) 859-6921

Larry A. Johnson
GTE
77 "A" Street
Needham, MA 02194
(617) 449-2000 ext. 3248
LJOHNSON@MIT-MULTICS

* Juern Juergens
SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02254
(617) 890-6900 ext. 316
JJURGENS@ECLB

Matt Kaufmann
Burroughs Corp.
Austin Research Center
12201 Technology Blvd.
Austin, TX 78727
(512) 258-2495
CMP.BARC@UTEXAS-20

Prof. Richard A. Kemmerer
Computer Science Department
University of California
Santa Barbara, CA 93106
(805) 961-4232
DICK@UCLA-CS

John C. Knight
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22903
(804) 924-1030
UVACS!JCK@SEISMO

Major Al Kopp
Ada Joint Program Office
OUSDRE/R&AT
Pentagon Room 3D139 (Fern Street)
Washington, DC 20301-3081
(202) 694-0211
AKOPP@ECLB

* Thomas M. Kraly
IBM Federal Systems Division
Software Eng. & Tech. 4D08
6600 Rockledge Drive
Bethesda, MD 20817
(301) 493-1449

Dr. Jack Kramer KRAMER@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2263

Eduardo Krell
3804 Boelter Hall
UCLA
Los Angeles, CA 90024

Kathy Kucheravy
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755

Dr. Kenneth Kung KKUNG@USC-ECLA
Hughes Aircraft Company
Ground Systems Group
M. S. 618/Q315
P. O. Box 3310
Fullerton, CA 92634
(714) 732-0262

* Carl Landwehr LANDWEHR@NRL-CSS
Code 7593
Naval Research Laboratory
Washington, DC 20375-5000
(202) 767-3381

* Mike Lake MLAKE@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Division
Alexandria, VA 22311
(703) 845-2519

Randall E. Leonard
Army Sys. Software Support Command
ATTN: ASB-QAA
Fort Belvoir, VA 22060

Nancy Leveson
ICS Department
University of California
Irvine, CA 92717
(714) 548-7525 or (714) 856-5517

Dr. Timothy E. Lindquist — LINDQUIS%ASU.CSNET@CSNET-RELAY
Computer Science Department
Arizona State University
Tempe, AZ 85287
(602) 965-2783

* Steven Litvintchouk — SDL@MITRE-BEDFORD
Mail Stop A180T
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7753

* David Luckham — LUCKHAM@SAIL
Stanford University
Computer Systems Lab, ERL 456
Stanford, CA 94305
(415) 497-1242

Dr. Glenn MacEwen
Computing and Information Science
Goodwin Hall
Queens University
Kingston, Ontario
K7L 3N6
(613) 547-2915 or (613) 548-4355

* Ann Marmor-Squires — MARMOR@ISI
TRW
Defense Systems Group
2751 Prosperity Avenue
Fairfax, VA 22031
(703) 876-8170

Eric Marshall — PAYTON@BBNG
System Development Corporation
P.O. Box 517
Paoli, PA 19301
(215) 648-7223

* Adrian R. D. Mathias — RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

* Terry Mayfield — TMAYFIELD@ECLB
Institute for Defense Analyses
Computer & Software Division
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2479

* John McHugh — MCHUGH@UTEXAS-20
Research Triangle Institute
Box 12194
Research Triangle Park, NC 27709
(919) 541-7327

Rudolf W. Meijer — RMEIJER@USC-ECLB
Commission of the European Communities
Info. Tech. and Telecomm. Task Force
A25 9/6A
Rue de la Loi 200
B-1049 Brussels, Belgium
PHONE: +32 2 235 7769

* Donn Milton — VRDXHQ!DRM1@SEISMO
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

* Warren Monroe — WMONROE@ECLA
Hughes Aircraft Co.
P.O. Box 3310
FU-618/Q315
Fullerton, CA 92634
(714) 732-2887

Mark Moriconi — MORICONI@SRI-CSL
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-5364

* LCDR Philip A. Myers — MYERS@NRL-CSR
Space and Naval Warfare Sys. Command
SPAWAR 8141A
Washington, DC 20363-5001
(202) 692-8484

* Karl Nyberg — NYBERG@ECLB
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

* Myron Obaranec — LAKSHMI@CECOM-1
U. S. Army, CECOM
Fort Monmouth, NJ 07703
ATTN: AMSEL-TCS-SIO
(201) 544-4962

Frank J. Oles
Thomas J. Watson Research Center
P.O. Box 218
88-K01 Route 134
Yorktown Heights, NY 10598
(914) 945-2012

Mahmoud Parsian
SDI Inc.
P. O. Box 4283
Falls Church, VA 22044

Diana B. Parton — DBP@MITRE-BEDFORD
The MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7754

* Don Peters
Comm. Sec. Establishment
Dept. of Nat. Defence
101 Colonel By Drive
Ottawa K1A OK2 CANADA
(613) 998-4519

* John Peterson — PETERSON@TYCHO
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755
(301) 859-6790

* Joseph E. Pfauntsch, MS 29A — JEP@FORD-COS4
Ford Aerospace & Comm. Corp.
10440 State Highway 83
Colorado Springs, Colorado 80908
(303) 594-1326

* Richard Platek — RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

Erhard Ploedereder
Tartan Labs
411 Melwood Avenue
Pittsburgh, PA 15213
(412) 621-2210
PLOEDEREDER@TARTAN

* David Preston
IITRI
5100 Forbes Blvd.
Lanham, MD 20706
(301) 459-3711
DPRESTON@ECLB

Sri Rajeev
AT&T Bell Laboratories
Room 1-342
190 River Road
Summit, NJ 07901
(201) 522-6330
IHNP4!ATTUNIX!RAJEEV@BERKELEY

* William D. Ricker
The MITRE Corporation
M/S K229
Burlington Road
Bedford, MA 01730
(617) 271-3001
WDR@MITRE-BEDFORD

* R. Max Robinson
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2097
RROBINSON@USC-ECLB

W. A. Robison
30 Charles Street West
Apt. # 1811
Toronto, Ontario, CANADA
M4Y 1R5
(416) 925-0751

* Clyde G. Roby
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2541
CROBY@ECLB

Ken Rowe
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755

John Rushby - EL393
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-5456

RUSHBY@SRI-CSL

* Mark Saaltink
I. P. Sharp Associates
265 Carling Avenue
Suite 600
Ottawa, Ontario, Canada K1S 2E1
(613) 236-9942

SAALTINK@MIT-MULTICS

Marvin Schaefer
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6880 or (301) 859-6818

SCHAEFER@USC-ISI

* Mike Schwartz
Mailstop L0402
Martin-Marietta
Denver Aerospace
P. O. Box 179
Denver, CO 80201
(303) 977-0421

UCBVAX!HPLABS!HAO!DENELCOR!
ALUVAX!MMADVAX!SCHWARTZ@BERKELEY

Dev Sen
STC IDEC LIMITED
Technology Division
Six Hills House
London Road
Stevenage
Hertfordshire S61 1YB ENGLAND
PHONE: 011-44-438-726161

* Jerry Shelton
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

VRDXHQ!JHS@SEISMO

* Brian Siritzky (212) 460-7239
Dept. of Computer Science
Courant Institue of Math. Sciences
New York University
251 Mercer Street
New York, NY 10012

SIRITZKY@NYU-ACF2 or
...CMCL2!ACF2!SIRITZKY

* Roger Smeaton — SMEATON@NOSC-TECR
NOSC, Code 423
San Diego, CA 92152
(619) 225-2083

Michael Smith — MKSMITH@UTEXAS
ICSCA
2100 Main Building
University of Texas
Austin, TX 78712
(512) 471-1901

* Ryan Stansifer — RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

* David Sutherland — RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

Steve Sutkowski — INCO@USC-ISID
Inco Inc.
8260 Greensboro Drive
McLean, VA 22102
(703) 883-4933

Michael Thompson
Astronautics Corporation of America
P. O. Box 523
Milwaukee, Wisconsin 53201-0523
(414) 447-8200

* Friedrich von Henke — VONHENKE@SRI-CSL
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-2560

Barry Watson — WATSON@ECLB
Ada Information Clearinghouse
IITRI
Room 3D139 (1211 Fern St., C-107)
The Pentagon
Washington, DC 20301
(703) 685-1477

Doug Weber
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020
RPLATEK@ECLB

* Steve Welke
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2393
SWELKE@ECLB

Col. William Whitaker
WIS/JPMO/ADT
7726 Old Springhouse Road
Washington, DC 20330-6600
(202) 285-5065
WWHITAKER@ECLB

* Jim Williams
MITRE Corporation
Mailstop B332
Burlington Road
Bedford, MA 01730
(617) 271-2647
JGW@MITRE-BEDFORD

Jim Wolfe
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2109
JWOLFE@ECLB

Larry Yelowitz
Ford Aerospace and Comm. Corp.
Western Development Lab. Div.
Mailstop X-20
3939 Fabian Way
Palo Alto, CA 94303
(415) 852-4198
KLY@FORD-WDL1

* Christine Youngblut
Advanced Software Methods, Inc.
17021 Sioux Lane
Gaithersburg, MD 20878
(301) 948-1989
CYOUNGBLUT@ECLB

* Margie Zuk
Mailstop B321, Bldg B
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7590

MMZ@MITRE-BEDFORD

APPENDIX B

Documentation from the European Efforts

The papers found in this Appendix were provided by the Dansk Datamatik Center (DDC). Since Kurt Hansen of DDC was unable to bring sufficient copies for all attendees, the Dansk Datamatik Center has allowed IDA to reproduce and include these documents as part of the Proceedings.

NOTES

NOTES

The Draft Formal Definition of Ada®

Commission of the European Communities: Multi-Annual Programme

Technical Annex

14 December, 1984, version 2

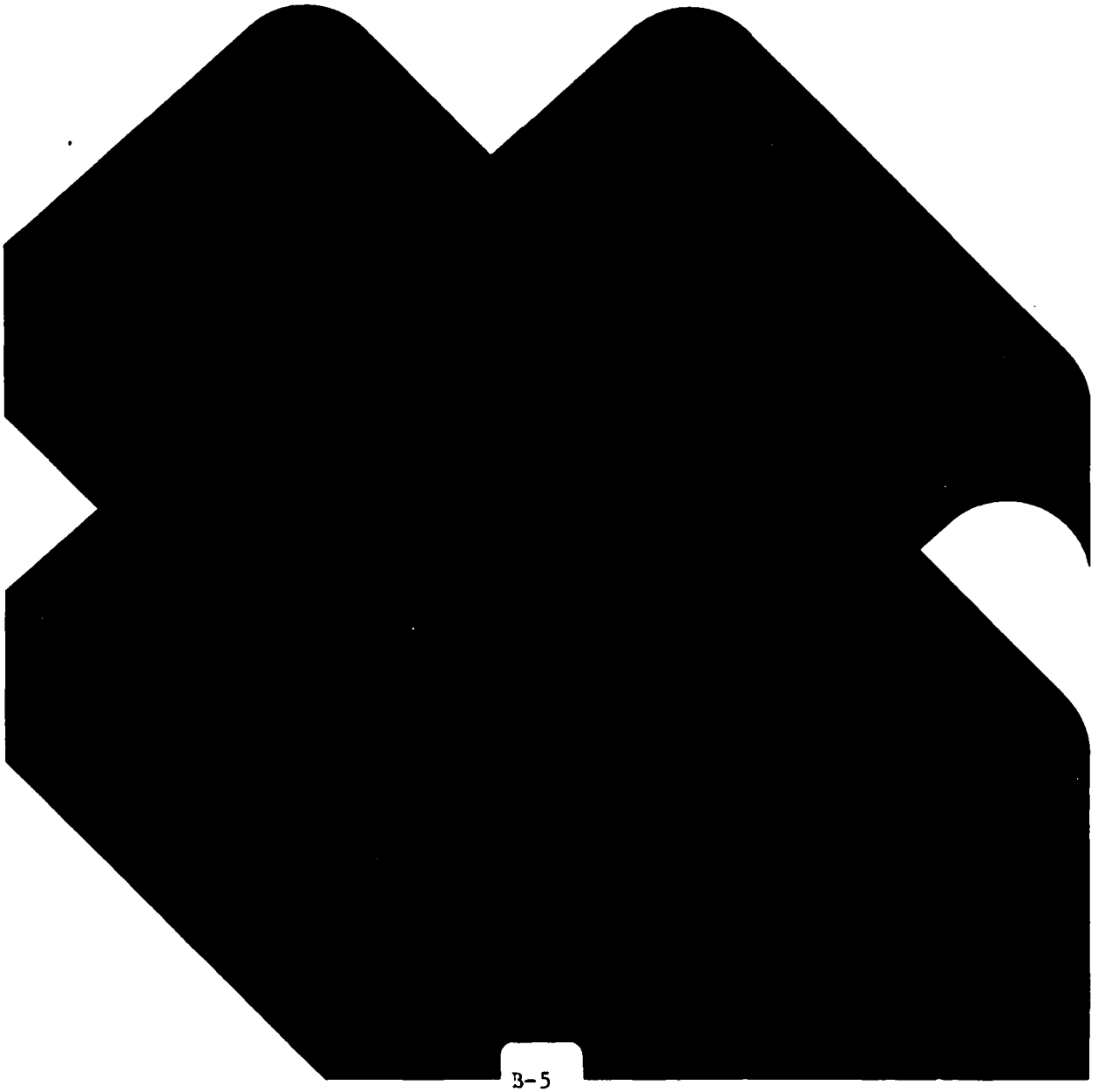
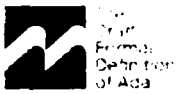


Table of Contents:



1. Project Title

The title of this project shall be:

"THE DRAFT FORMAL DEFINITION OF ANSI/MIL-STD 1815A Ada"

-- hereafter referred to as the "Ada FD".

2. Project Summary

The project aims at developing the draft Ada language formal definition, the Ada FD.

The task will be completed using state-of-the-art techniques in formal specification methods. Different specification approaches will be carefully studied, and the most promising methods will be chosen.

Work Packages: The project is foreseen to progress as follows:

- D - A "difficult" subset of Ada will be selected,
- C - a set of combinable specification techniques adequate for the definition of full Ada will be tentatively selected, and
- E - a trial definition will be developed.
- F - The trial definition will be evaluated, and on this basis
- IJ KL - a full scale draft Ada formal definition (the Ada FD) will be developed.
- P - In parallel, annotations of the Ada FD will be developed.
- R - Extensive cross referencing to the Ada standard document (ANSI/MIL-STD 1815A) will be developed.
- V - The work on the Ada FD, it's annotation, and correlation to existing reference manuals will be reviewed on a regular basis.
- NO - Tools for manipulating the Ada FD will be developed.
- Q - mappings from the proposed Ada FD to the NYU SETL interpreter for Ada, will be documented, as will
- S - a study of the feasibility of automated verification of the ACVC test suite with respect to the Ada FD.
- X - Finally Educational Issues will be addressed.

Objectives

It must be emphasized that the completed Ada FD will define the Ada language as found in ANSI/MIL-STD 1815A (Revision January 1983). Whereever this latter might be inconsistent, incomplete or ambiguous, the produced Ada FD will leave the subject undefined.

3. Objectives

The main objectives of this project are:

- To obtain as concise a definition of the full ANSI Ada language as is today feasible, in a form which

  (0) may serve as a reference for questions on Ada,

  and is suitable for further research on the following topics:

  (1) formal work in the areas of proof systems for Ada programs,

  (2) correct development of correct Ada interpreters and compilers,

  (3) the meaningful generation and verification of Ada test programs, incl. validation of the ACVC test suite, and

  (4) the derivation of informal, but precise, unambiguous Ada reference manuals for various user groups,

  in order to help provide:

  (5) input to the ongoing standardization work on Ada, in particular to support the ISO future review of the Ada standard, and

  (6) a worthy, broad, and commonly accepted candidate for the formal definition component of a future Ada ISO Standard.

- And to further the propagation of Ada, as well as teaching professionals how to read, understand and use an Ada FD in their present position.

Subsidiary objectives are:

- To help unite various approaches to the informal, and semi-formal descriptions of Ada (by studying, how to relate the proposed Ada FD to e.g. the NYU SETL interpreter for Ada)

- To further develop and research engineering methods suitable for the precise definition of large, complex software systems (by calling on a wide community of computer scientists to take part both in the actual Ada FD development, and its review), and thereby

- To further propagate the use of formal methods in software engineering.

4. Aims and Objectives with Respect to the Multiannual Programme

The following is quoted from TF-TIT/2472/84-EN rev. 3, start pg.36:

"3. Formal Definition of Ada

3.1. Background

Work on a Formal Definition (FD) of Ada is of prime importance for the rigour and stability of the Ada Standard. Eventually, a completely formal description could be the prime form of any programming language standard, with a narrative definition and validation test suite as complements. However, even though the main mathematical formalisms to cover the important aspects are probably avai - lable, combining them effectively and applying them to the concrete case of a language as comprehensive as Ada is a matter which still needs development. Part of this work will be for tools that help to make the description more tractable, and hence more usable for a number of purposes: not only as a candidate for the ultimate language standard, but also as a basis for derivation of correct compilers, and for reasoning about properties of Ada programs. Another aspect is that of making the description executable, so that it would be used to process the Ada validation test suite, and Ada programs in general.

Work on a FD of Ada cannot proceed in isolation: it needs to recognize first of all the existiang standardization effort and their revision cycle. The work of ISO TC97/ SC5/WG 14 "Ada" has just begun (first meeting 10-11 April 1984). At the first meeting it was confirmed that the basis for the initial ISO standard shall be the Ada Reference Manual, and that a formal description is not considered at the stage. In fact a separate working group ISO TC97/SC5/WG 16 "Guidelines for the development of standards within SC5" may at some stage address the usage of a formal description for standardization of programming languages. Thus any FD project should at least establish liaison with WG 14 and WG 16. Other standards liaison, e.g. with ANSI and ECMA, may also be useful.

There is a possibility that the US will fund some work on the same subject. In that case a collaboration could be envisaged, most likely in the form of independently funded, but complementary projects, which have a large measure of mutual cognizance.

3.2 Guidelines for the Formal Definition Project(s)

The following guidelines will apply to any project proposal under this heading. They are for a large part based on the advice given by the Ada-Europe working group on Formal Semantics of Ada, which has held intensive discussions on this subject over the past one and a half years."

The above quoted section is in close harmony with what the proposers of this project believe.

In order to show that the project complies with the aims and objectives of the multiannual programme, we have numbered and quoted the EEC requirements below -- together with our plans on how to fulfill them.

1. "All proposals shall contain details explaining on what basis and to what extent the approach(es) put forward can be considered "formal"."

Definitions can be expressed in various styles:

- Systematic: The gross outlines of a 'formal' specification method is followed -- using some informally explained specification language(s),
- Rigorous: and certain, or all relevant, but not necessarily all aspects of, properties of this language and of the constructed specification are 'formally' expressed,
- Formal: and 'formally' verified or defined.

In the previous three paragraphs the word 'formal' has been used in the sense it is used in mathematical logic.

It is here tentatively being proposed to split the Ada FD into basically three parts:

- Static Semantics: dealing with all the statically decidable properties that any Ada program must satisfy, and which a compiler is specified to check.
- Dynamic Sequential Semantics: dealing with the run-time, action, or execution semantics of all but the tasking aspects of Ada.
- Dynamic Parallel Semantics: dealing almost exclusively with the time-dependent, and tasking aspects of Ada.

Aims and Objectives with Respect to the
Multiannual Programme

This split has been chosen for pragmatic reasons, and is motivated below.

It is further being tentatively proposed to define:

- deterministic aspects of Ada denotationally,
- non-deterministic, but not concurrent, aspects of Ada axiomatically/algebraically, and
- concurrent aspects of Ada, i.e. Ada tasking, structural operationally.

In addition we may find it desirable to express certain absolute, or relative, partially ordered, time-dependent features of Ada using temporal, or interval logic.

For the denotational semantics we propose to choose, as our departure point for a fully, formally definable specification language, that of VDMs META-IV, but with additions and restrictions, henceforth referred to as ML4.

In the static semantics a simple, applicative subset of ML4 will be proposed, and the definition will be a standard, denotational semantics (non-exit, non-continuation style) model. Thus the static semantics model will be fully formal.

For the greater parts of the dynamic sequential semantics an imperative version of ML4, using the so-called exit mechanism, will be proposed, and the definition will be a denotational model which can be fully, denotationally, i.e. formally explained. We propose to "decorate" the applicative ML4 with imperative-looking combinators like statements, sequencing, and exit constructs, in order to render the definition more readable. It should be noted that the "imperative" combinators are but a well-disciplined precursor to the "abstract semantic algebras" of e.g. Peter Mosses. In this sense our dynamic semantics definition of Ada is fully formal.

The storage model of Ada: values, locations (pointers), allocation, assignment, and contents-taking, will be proposed expressed in a style reminiscent of the CLEAR or ASL algebraic semantics specification language. Other, minor parts of "sequential" Ada may likewise be, and in cases, alternatively, rather than only exclusively, algebraically defined. To the extent, these metalanguages are formal and combineable this definition will be formal.

The definition of Ada tasking is here being proposed to be defined using the SMoLCS derivative of structural operational semantics.

Since SMoLCS can be expressed in an algebraic style, using ASL, it turns out that the definition of Ada tasking can be made technically similar to the algebraic style mentioned above.

It will finally be attempted to give the combination of the 4-5 specification parts a formal explanation. This may be done either "absolutely" (ideally): with respect to the underlying specification languages, or "relatively": with respect to the actual, resulting Ada FD. To the extent that this can be expressed formally, the whole Ada FD is formal. To the extent it cannot be properly formalized, the Ada FD is only rigorous. We believe that it is feasible to express the "relative" meaning of combining the specification parts.

2. "Review procedures shall be incorporated in that workplan as an integral part of the effort, in order to promote acceptance of the results; the problem of liaison to the User Community shall be addressed."

A document: "The Rôle of the Ada FD" will be proposed. It will define the uses and user groups of the Ada FD. On the basis of such an approved document a suitably large list of representative users from each of the groups, and from Europe and the US, will be established. The user groups will review the ongoing work in two forms: write-in reviews in response to broadcast mailed reports, and meeting reviews where the Ada FD project partners present their ongoing work. An Ada FD review board, set up independently by the CEC, will negotiate with the presently proposed project partners on any discrepancies there might arise. None, of the above mentioned reviews are funded by this project, except for contractors part. It is also pointed out, that the review is essential, but it is the responsibility of the contractors, to formulate their further actions in view of the review outcome.

3. "The FD shall base itself on the results of existing work as far as possible; this includes the incomplete (out-of-date) descriptions by INRIA (F), and DDC (DK); the work at NYU -- SETL (US); as well as the Karlsruhe attribute grammar (D)."

The work will start from scratch, but based on the current state of the art, both in formal methods and in Ada formal definition work.

The main contractor of the project has completed a rigorous definition of Ada using the VDM approach, and intends to build, not only on that work, but on some of the people who did it.

It is also included to study rigorous analyses and mappings

from the Ada FD to the NYU SETL interpreter for Ada.

Since the INRIA work is basically using the same denotational approach as will the presently proposed Ada FD, one can say that it will also incorporate the INRIA work. But since this latter reflects a rather early attempt which did not define anyway near the full Ada (minus tasking and storage), and at a stage where Ada was rather different from what it is now, one may claim that we are not proposing any explicit mapping from the proposed Ada FD to the INRIA work.

4. "The FD shall be developed using reasonably few and concise methods, which shall be uniformly applied to the whole language. The theoretical foundations for the combination of several methods shall be given, and proof and verification theories for the FD shall be developed."

We refer to the remarks made in connection with point 1 above. One may claim that the proposed number of different specification methods does not satisfy the "few" criterion. It may certainly be possible, but, it is felt, not entirely desirable, to cut down on the number of different methods. First we could, e.g. give constructive, denotational models for storage and the other nondeterministic features of Ada -- and that should indeed be considered. Secondly one could, both theoretically, and practically, express all of Ada in one style, using either of e.g. de Bakkers, Tochers, or Plotkins specification methods. This would solve the "combination" problem, but not the accessability (readability, and conciseness) problem. We therefore maintain the presently proposed approach.

5. "The FD shall not be unduly constrained by the necessity to describe certain concepts like representation clauses, implementation defined attributes, and some pragmas. However, all possible effort shall be made to integrate these concepts."

An attempt to express some of these aspects will be made, and it is here suggested to do so axiomatically -- and orthogonally to the remaining, complete and consistent Ada FD.

6. "The FD document shall be coordinated/integrated with the existing Reference Manual."

This is a very important point, and is described more detailed in the description of work package R, pg. I-35.

7. "The FD shall be the source of derived documents for

a variety of user interests and needs; for example, the FD shall be suitable for the verification of proof rules for Ada programs.

Work packages P and R, pp. 33 & 35 outline our proposal in this area. We tentatively define three groups of users of such documents:

- Ada text book and reference manual writers, and Ada language educators and teachers -- and, through them, ordinary Ada programmers,
- Ada programmers interested in proving their Ada programs correct, and
- Ada compiler and interpreter implementors.

For the first group derived documents should describe Ada in natural language terms, in a tersely, and Ada FD related manner. See work package P, pg I-33, for more details.

For the second group the derived documents should consist of informally annotated, formal proof rules, and preferably guide lines on their use.

For the latter group a derived document could outline the methods that can be used to derive correct interpreters and compilers from the Ada FD. Since the literature, by now, is fairly full of such information this will not be proposed done in this project.

8. "The FD shall be suitable for the validation of the ACVC test suite. An effort shall be made to provide means for mechanically testing the ACVC against the FD (e.g. by having an executable FD, or making an executable version automatically derived from the FD by a tool)."

In this project alternative approaches will be studied:

- indirect executability, as above, via studies of mappings to the NYU SETL interpreter. Work packages Q pg 34, will study this aspect.
- proof of the ACVC program incorrectness/correctness Work package S, pg I-36, will study this aspect.

9. "The development of the FD requires support tools to manipulate the FD document and to coordinate it with the Reference Manual."

This task is taken care of by work packages N and O, pp I-31-32.

10. "All tools will be developed as (M)APSE tools."

Yes.

5. Current State of the Art

It is widely recognized that software engineering, unlike the more established engineering disciplines, is still largely at the craft stage in that the techniques in common use lack an underlying scientific basis. In particular, the early stages of the system life cycle (requirements analysis, specification and high-level design) are rarely treated in a disciplined way by the software engineer. Yet, these stages are worthy of particular attention since faults generated here have been shown to be the most difficult to detect and the most costly to repair. The growing awareness of these problems has led to the development of formal specification and systematic development methods based upon recent advances in mathematics and computer science.

In recent years there has been intensive research and development of a variety of approaches to formal specification and systematic program development in a number of centres, principally in Europe and North America. A large number of real and laboratory applications have by now been carried out and, at least for non-concurrent aspects of systems, a consensus seems to be emerging regarding the desirable characteristics of such approaches. Experimental toolsets to support these approaches have also been developed and used on real projects.

An ESPRIT preparatory study has been carried out in this area by the Dansk Datamatik Center (DDC) and Standard Telecommunications Laboratories Ltd (STL). The report of this study is in two parts. The first part is a broad survey of the state of the art in formal development theories, methods and tools, comparing the situation in Europe, the U.S.A. and Japan. The second part is an in depth evaluation of one particularly well-established method, VDM. This study provides probably the most extensive and up-to-date view of the field addressed in this proposal, but other useful surveys of development methods in general (not just formal methods) are available, for example, the DOI Study of Ada-based System Development Methodology the 'Methodman' document for Ada and the survey of Software Tools for Application to Large Real-time Systems (the 'STARTS' guide).

In the past, the principal approaches to formal methods have been characterised as "model-oriented" or "property-oriented". In the model-oriented approach, specifications and designs are explicit models of systems constructed from well-defined primitives. In the property-oriented approach, specifications are given in terms of axioms defining only the relationships of operations to each other (as in, for example, the so-called "algebraic" approach).

Important centres of research and application in the model-oriented school include the Dansk Datamatik Center, the University of Manchester and Standard Telecommunication Laboratories (for VDM), SRI International (for HDM), USC Institute of Information Sciences (the GIST project), the University of Oxford (for Z) and Higher Order Software Inc. (for the HOS method).

Important centres for the property-based approach include the University of Edinburgh (Clear), SRI International (OBJ, CLEAR), MIT and Xerox PARC (Larch), the Universities of Pisa and Genoa, the Technical University of Munich (CIP), USC Institute of Information Sciences (for Affirm), the Technical University of Berlin, and the University of Passau.

It is notable that the two schools now recognise attractive benefits in each other's approaches and systems which attempt to provide the benefits of both are increasingly being proposed. Such ideas are evident at, for example, MIT, Oxford, Manchester, DDC, Xerox PARC, STL and SRI.

In the area of concurrency there is much less agreement on the "right" approach and a large number of contrasting theories are being researched. These include algebraic approaches (e.g. CSP from Oxford and CCS from Edinburgh University), net theory (GMD Bonn), temporal and modal logics (Manchester University, SRI, Stanford, etc.,) and label-event and SMoLCS systems (Pisa and Genoa). In September 1983, a workshop organised jointly by the U.K. Science and Engineering Research Council and Standard Telecommunication Laboratories, STL, was held in Cambridge (U.K.) at which many of the leading researchers in the field were present and the principal approaches compared. The forthcoming published proceedings will provide valuable input for this proposed project.

A number of attempts have been made to support some concurrency features alongside established methods for sequential systems - for example CSP with VDM (at DDC), temporal logic with HDM (at SRI), the rely/guarantee condition extensions to VDM (at Manchester) and predicate-transition nets (at GMD). An ESPRIT pilot project (the GRASPIN project) is attempting to utilise Petri nets and axiomatic abstract data types in a coherent framework. However, in general, combining various approaches based on differing semantic theories raises fundamentally difficult problems; the issues involved in this were explored in a NATO-sponsored workshop organised by the Dansk Datamatik Center in May 1984. It was attended my many of the leading experts on semantics. The proceedings of this workshop will clearly provide valuable input to the proposed project.

A number of formal approaches have been supported by experimental toolsets, some of which have been utilised in real-world projects. Notable efforts have been developed at USC-ISI (Affirm), the University of Texas (Gypsy) and HOS Inc.(Use-it). Database systems for specifications have been explored at Xerox PARC (PIE). Notable work in theorem proving has been carried out at SRI (Boyer-Moore), the University of Nancy (Reve) and the University of Edinburgh (LCF), among other centres. Significant programming environment efforts have been carried out at INRIA (Mentor), CMU (Gandalf) and in Japan (Iota).

It must be noted, however, that most of these toolsets are experimental vehicles and could not be utilised directly in industrial situations. (Exceptions are Use-it, marketed commercially by HOS, and possibly Gypsy.) Considerable work is required to develop tools capable of handling large-scale industrial applications. It will clearly be necessary to develop full scale database-oriented programming environments based around formal methods. This highlights a gulf between researchers and practitioners which must be bridged for any method: the promising ideas emerging from research must be proven in industrialscale case studies and packaged for transfer and use in an industrial context. Relatively few 'methods' have yet reached this stage of maturity which would be characterised by the availability of significant published case studies, textbooks and industrially oriented training courses. (VDM is one of the most mature according to these criteria.)

In terms of applications, the more established approaches have been used on a significant number of real-world projects. There appear to have been more of these in the U.S.A. HDM, for example, has been used to specify and prove security on a number of operating system kernels (KSOS, PSOS and SIFT). HOS has been used on a number of embedded military systems. Affirm has been used to specify and prove a security kernel, various communication protocols and a military message switch. Gypsy has similarly been used for message switching and for part of an aircraft control system.

In Europe, the most widely used formal method in industrial situations is probably VDM. VDM has been applied to a variety of projects in a number of countries: Austria, Denmark, the Federal Republic of Germany, the United Kingdom and Italy. Applications include the development of compilers, database systems, aspects of operating systems, and office automation systems.

These formal systems have been tried out in various applications, among these is Ada. These studies contain attribute grammar definitions of Ada (Karlsruhe), incon-

plete Ada (INRIA), DIANA-syntax, SETL executeable description (NYU), and the somewhat outdated DDC Ada FD. The latter is the basis from which the DDC validated Ada compiler is derived.

The summary above of formal methods and Ada definition will form a very strong base for a development of an Ada FD. Furthermore, current research will be incorporated into the project - specially the ESPRIT funded RAISE (Rigorous Approach to Industrial Software Engineering) seems to be able to contribute considerably.

6. Project Description

6.1 Overview

This section provides an overview of the contents of this proposal, and includes an overview of the deliverables.

The proposed Ada FD project may be seen to consist of five major categories of work:

- Selection of appropriate, "difficult" example subset of Ada (xAda), selection of appropriate formalisms to be used in a FD of xAda, and the trial FD of xAda -- all this intended for review and approval of general approach.

- The actual draft FD of ANSI/MIL-STD 1815A Ada.

- The derivation of a natural language description of Ada from the FD, and their correlation to the existing ANSI Ada Reference Manual(s); this category also includes liason with appropriate standards organisations: ANSI, ISO, ECMA, etc. as well as preparation of educational type of documentation.

- Development of new, and adapting existing tools for the manipulation of the Ada FD; and feasibility study of ACVC test suite validation from the Ada FD.

- Review of Ada FD and informal, natural language descriptions and correlations -- to be held at regular intervals throughout the project.

The main deliverables will be:

- A draft Formal Definition (FD) of ANSI/MIL-STD 1815A Ada to the extent, that the standard is unambiguous and complete. Exhaustive annotations, and correlations to existing informal reference manuals will be made.

- Evaluation reports arising from regular reviews.

- Report stating the results of the study of ACVC validation feasibility.

- Tools, written in Ada, and supported by an APSE, for handling the Ada FD, and prepared for the introduction of possible proof systems and ACVC validation.

6.2 Work Packages and their Interrelation

This section contains a detailed description of the project in terms of relevant work packages.

Work Package

Identification: A

Name: Start up

Purpose: Project initialization and liaison

Contents:

- Set up of tools, equipment, files etc. necessary for project management
- Establishment of contacts to other groups working in the area (ANSI, ISO, ECMA, ...).
- Construction of mailing lists and opening letters to potential reviewers and users of an Ada FD.

Requisites: pre: None
post: WPs C-D

Man Months: 1

Deliverables: Report 1: Project procedures
Report 2: Review Procedures
Report 3: Review Groups (Mailing Lists)

(|) Review: Reports 2-3

(|) The reviews mentioned in this and the following work packages are external reviews, and is done via work package V, pg I-39.

Work Package

Identification: B

Name: The Rôle of the FD (Formal Definition)

Purpose: Defines the requirements to be fulfilled by a FD of Ada -- identifies the various uses such a FD may have.

Contents:

This deliverable defines the various user groups of an Ada FD (incl. possible Proofs Systems for the Ada FD), and the uses these groups may have of such a FD. Roughly speaking the groups include (1) Ada programming language reference manual writers (and, through them, Ada programmers), (2) Teachers of Ada programming, (3) Ada interpreter and compiler developers, (4) APSE developers, (5) Computer scientists interested in studying Ada related matters (such as e.g. proof systems, formal validation, formal specification, etc.), and (6) International and national Ada language standardization organisation members.

Requisites: pre: None
post: WP E-X

Man Months: 1

Deliverables: Report 4: The Rôle of the FD of Ada

Review: Report 4

Work Package

Identification: C

Name: Tentative Specification Language

Purpose: Select a tentative specification language for a "difficult", example subset Ada (WP D), the specification of which (WP E), can serve as a basis for reviews and subsequent approvals.

Contents:

This WP will tentatively select the specification techniques to be used for the full FD of Ada. It is to be expected that these might include:

- Denotational semantics techniques for the specification of the (sequential) deterministic aspects of Ada,
- Algebraic semantics techniques for the specification of the non-deterministic (non-concurrent) aspects of Ada,
- Structural operational semantics (labelled event system) techniques for the specification of concurrent aspects of Ada, following the ASL-SMoLCS approach also this part can be expressed in an algebraic style, and possibly
- Temporal (or interval) logic techniques for the specification of temporal (time) aspects of Ada.

The chosen techniques will represent the main streams of established, international research in the area of specification techniques.

Requisites: pre: WP A
post: WP E

Man Months: 5

Deliverables: Report 5: Informal Description of Trial Specification Languages.

Review: By WP F: Report 5

Project Description

Work Package

Identification: D

Name: Example Ada subset selection

Purpose: This "difficult" Ada subset shall serve as the basis for a trial FD, see WP E.

Contents:

A representative, but specification-wise "difficult" subset of Ada is to be selected -- a subset illustrating all relevant aspects of Ada, ie such which examplifies deterministic, as well as non-deterministic; sequential, as well as tasking; time-independent, as well as time-dependent; static as well as dynamic semantics; syntactic, semantic, and pragmatic aspects of Ada, and thereby also the complexity of Ada.

Requisites: pre: WP A
post: WP E-X

Man Months: 4

Deliverables: Report 6: Example "Difficult" Subset Ada

Review: By WP F: Report 6

Project Description

Work Package

Identification: E

Name: FD of a "Difficult" example Ada subset.

Purpose: To show the feasibility, and appropriateness of the chosen formal specification method.

Contents:

A definition of the "difficult" example Ada subset, together with exerpts of an informal, natural language annotation of same, and its correlation to the ANSI informal reference manual.

The actual work will be done iteratively. 3 persons will work simultaneously on up to three aspects of the Ada language (deterministic, non-deterministic, and tasking). These three persons will submit early attempts, sketches, drafts, for international review in order to guarantee approval.

Requisites: pre: WPs B-C-D
post: WPs F-N

Man Months: 12

Deliverables: Report 7: Formal Definition of "Difficult", Example Ada Subset

Review: See WP F

Work Package

Identification: F

Name: Initial Review and Approval

Purpose: To set the stage for the full, FD of Ada, by assuring that the chosen method is acceptable.

Contents:

International advisory groups review the FD of the "difficult", example Ada subset (see also workpackage V) its derived natural language explication, and its correlation to the ANSI informal description.

This work package (F) is separate from work package V, pg I-39, the general, ongoing review of ongoing Ada FD activities.

Funding of this activity is not included in this project, except for the contractor part.

Requisites: pre: WPs B-C-E
post: WP G

Man Months: 1

Deliverables: Report 8: Review of WPs B-C-E, conclusions and propose further actions.

Review: No

Project Description

Work Package

Identification: G

Name: Final Specification Language

Purpose: Serves as basis (input) for WPs H-I-J-K-L (Ada FD) and WPs N-O (Ada FD Tools).

Contents:

A complete description of the full set of formal specification languages used in the resulting Ada FD. This work consists of individual work of the specific denotational, algebraic, structural operational, and other, semantic specification notations, as well as on the possibility of their combined semantics.

We refer to remarks made in the contents section of WP C.

Requisites: pre: WP F
post: WPs H-I-J-K-L-N-O-X

Man Months: 7

Deliverables: Report 9: Final Specification Languages and Methods -- a description of the individual and combined semantics of the chosen specification languages and methods.

Review: Report 9

Project Description

Work Package

Identification: H

Name: Ground rules for natural language explication

Purpose: To establish rules for the informal, natural language explication of formal definition formulas, and for the correlation to existing Ada reference manuals.

Contents:

Identification of rules for deriving natural language descriptions, or explications (explanations) of the Ada FD formula, and for the systematic correlation of the Ada FD to the existing Ada reference manuals. This work is concerned with "style". The target, natural language will be english.

Requisites: pre: WP G
post: WPs P-R

Man Months: 1

Deliverables: Report 10: Guidelines on Ada FD Explication
Report 11: Guidelines on ANSI/MIL-STD 1815A Ada FD Correlation

Review: Reports 10-11.

Project Description

Work Package

Identification: I

Name: Formal Definition of Ada Static Semantics

Purpose: To establish a concise, formal definition of all the statically decidable properties of any Ada program.

Contents:

Two issues will be addressed:

- The design of an abstract syntax and a correlated concrete syntax for the Ada language.
- The formal definition of the static semantics of Ada using the formalism chosen in WP G, with respect to the ANSI/MIL - STD 1815A Ada standard.

The work will be carried out in two phases of approximately equal lengths. The first phase results in a draft proposal subject to an intermediate review. The second phase ends with a review approved FD of Ada static semantics.

Requisites: pre: WPs G-R
post: WPs L-O-P-Q-R-S-X

Man Months: 12

Deliverables: Report 12: The Concrete and Abstract Syntax of Static Ada, and their Mutual Translations
Report 13: The Formal Definition of Ada Static Semantics.

These deliverables will be issued in two versions:

Reports ½12-½13: half-way, incomplete draft
Reports 12-13: final draft

Review: Reports ½12, ½13, 12, 13

Project Description

Work Package

Identification: J

Name: Formal Definition of Ada Dynamic Sequential Semantics

Purpose: To establish a concise, formal definition of the dynamic semantics of sequential and non-deterministic (but not tasking) aspects of the Ada language.

Contents:

Three issues will be addressed:

- Design of an abstract syntax suitable for expressing the dynamic semantics of Ada -- possibly correlated to the DIANA intermediate language.
- A correlator to (translator from) the static semantics abstract syntax language.
- A formal definition of the non-tasking aspects of the Ada language. This part may involve use of up to two specification languages, a denotational for the deterministic sequential aspects of Ada, and an algebraic for the nondeterministic, exclusive of tasking, aspects of Ada.

The work will be carried out in two phases of approximately equal lengths. The first phase results in a draft proposal subject to an intermediate review. The last phase ends with a review-approved FD of Ada dynamic sequential and non-determinstc semantics.

Requisites: pre: WP G
post: WPs H-L-O-P-Q-R-X

Man Months: 8

Deliverables: Report 14: Abstract Syntax for Dynamic Ada and a Translator from Static to Dynamic Ada Abstract Syntaxes.
Report 15: The Formal Definition of Ada Dynamic Sequential Semantics

These deliverables will be issued in two versions:

Reports ½14-½15: half-way, incomplete draft
Reports 14-15: final draft

Review: Reports ½14, ½15, 14, 15

Work Package

Identification: K

Name: Formal Definition of the Ada Dynamic Parallel (ie Tasking) Semantics

Purpose: To obtain a concise, formal definition of all the tasking, ie concurrent and time-dependent aspects of the Ada language.

Contents:

Only one issue will be addressed:

- The formal definition of the tasking and time-dependent aspects of the ANSI/MIL - STD 1815A Ada language. The word formal means: to the extent, that the metalanguages used can be combined formally.

The work will be carried out in two phases, as for WPs I-J.

Requisites: pre: WP G-R
post: WPs L-O-P-Q-R-S-X

Man Months: 12

Deliverables: Report 16: The Formal Definition of Ada Dynamic Tasking Semantics.

This deliverable will be issued in two versions:

Reports ½16: half-way, incomplete draft
Reports 16: final draft

Review: Reports ½16, 16

Work Package

Identification: L

Name: Integration of Ada Formal Definitions

Purpose: To combine the three part Ada formal definition (as obtained in WPs I-J-K) into one coherent, consistent, and complete formal definition (formal, as defined on page I-10) -- one which is suitably cross-referenced, indexed and otherwise checked.

Contents:

Three consistency and completeness issues will be addressed:

- Syntactic: among definition parts with respect to usage of abstract syntax defined domains and function types.
- Semantic: between definition parts with respect to pre/post conditions of defined functions, whether putatively defined, as in e.g. denotational definitions, axiomatically defined, as in algebraic definitions, or rewrite rule defined, as in structural operational definitions, etc.
- Pragmatic: between the FD and the informal Ada reference manuals.

The Correlation of the Ada FD to the ANSI/MIL-STD 1815A will have as its ANSI/MIL-STD 1815A component a document which is divided into a number of chapters, "one per group of language features". This integration work package will collect the appropriate parts from reports 12-13-14-15-16 (by means of the Ada FD Tool set) in a form analogous to the ANSI/MIL-STD 1815A layout.

Requisites: pre: WPs I-J-K
post: WPs I-J-K-O-P-Q-R-S-X

Man Months: 8

Deliverables: Report 17: The Formal Definition of Ada

Review: Report 17

Work Package

Identification: N

Name: Requirements for an Ada FD Tool set

Purpose: To establish the requirements that different Ada FD user groups will put on a set of software tools relating to the Ada FD.

Contents:

A number of portable, APSE-based software tools for the creation, maintenance and diverse uses of the Ada FD can be envisaged:

- editors: line, full-screen, and syntax-directed
- a variety of pretty printers/displayers
- Ada FD syntax and type checkers, ie not checkers of the syntax of Ada, but of the syntaxes of the Ada FD, and the function types of its defined functions.
- interfaces to possible Ada FD interpreters
- interfaces to possible Ada FD based proof/verification sub-system
- interfaces to possible Ada FD based ACVC test suite validators

In this work package a set of requirements are established for such a tool set.

Requisites: pre: WP E (L)
post: WP O

Man Months: 5

Deliverables: Report 19: Requirements for a Portable, APSE-based Ada FD tool set

Review: No

Project Description

Work Package

Identification: P

Name: Informal Explication

Purpose: To provide an english, ie. natural language explanation of the Ada FD.

Contents:

The Ada FD is necessarily terse, and expressed in a formal, symbolic language. To facilitate its reading, and hence its acceptance and use, it is proposed that the Ada FD be extensively annotated, in an english language, natural style.

It is expected that different user (target) groups will require different style explications -- the requirements for these will be defined in WP B.

This work will be done with respect to (wrt) the individual formal definitions -- as developed in WPs I-J-K, rather than wrt. the integrated Ada FD of WP L.

Requisites: pre: WPs H-I-J-K-L
post: None

Man Months: 6

Deliverables:
Report 25: An Informal Explication of the Ada FD -- an Introduction
Report 26: An Informal Explication of the Ada FD Static Semantics
Report 27: An Informal Explication of the Ada FD Dynamic Sequential Semantics
Report 28: An Informal Explication of the Ada FD Dynamic Tasking Semantics
Report 29: An Informal Explication of the Ada FD Combined Semantics

Review: Reports 25-26-27-28-29

Project Description

Work Package

Identification: 0

Name: Tool set Construction

Purpose: To create a portable set of APSE based tools suitable for a wide group of Ada FD developers and users.

Contents:

This work package consists of:

- The FD of the architecture of an Ada FD tool set
- The design of such a tool set
- The coding of such a tool set

We refer to the contents description for WP N.

The present work package will deal with the specific issues of the Ada FD: i.e. those for which the tools specifically know that the object to which they are applied is the Ada FD.

This is in contrast to tool sets that might have been developed for (ancestors of) the specification languages (META-IV, ML4, SMoLCS, CLEAR/OBJ, ASL, etc.) used in this project. Insofar as such (ie these latter) tools exists, this project will adapt them to the Ada FD tool set, thereby enlarging its scope and utility.

It is to be expected that certain tools already developed by the contractors go into the above tool set.

Requisites: pre: (WPs G-I-J-K-L-N)
post: None

Man Months: 19

Deliverables: Report 20: Ada FD Tool set: Architecture.
Report 21: Ada FD Tool set: Design.
Report 22: Ada FD Tool set: Users Manual
Report 23: Ada FD Tool set: Installation
Report 24: Ada FD Tool set: Primer

Software: Portable, APSE-based Ada FD tool set.

Review: No

Project Description

Work Package

Identification: Q

Name: Feasibility study: Mapping to the NYU SETL Ada Interpreter.

Purpose: To study the extent to which the Ada FD of this project may be correlated to the existing SETL programmed interpreter for Ada as developed by the New York University.

Contents:

There are two semi-formal, near- or fully executable models of Ada: the Karlsruhe (FRG) University Extended Attribute Grammar (EAG) description of Ada, and the New York University (NYU) SETL program interpreter for Ada.

In order validate to Ada FD, and in order to investigate the possibility of letting either of these descriptions serve as a basis for the ACVC test suite validation it is necessary to establish, reasonably formally, a "mapping" from (i.e. a correlation of) the Ada FD of this project, to either or both these descriptions.

This workpackage will study a possible mapping to the NYU SETL Definition.

Requisites: pre: WPS: Q-I-J-K-L

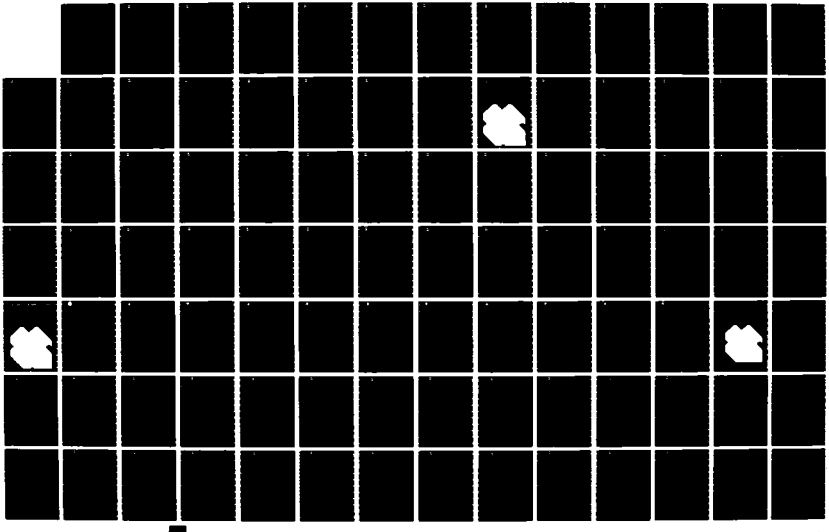
post: None

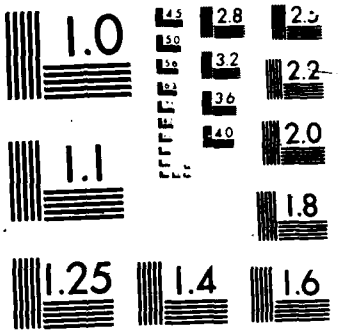
Man Months: 2

Deliverables: Report 30: Feasibility of a Mapping from the Ada FD to the NYU SETL Interpreter for Ada

Review: No

AD-A172 747 PROCEEDINGS OF THE IDA (INSTITUTE FOR DEFENSE ANALYSES) 3/4
WORKSHOP ON FORMA..(U) INSTITUTE FOR DEFENSE ANALYSES
ALEXANDRIA VA W T MAYFIELD ET AL. NOV 85 IDA-M-135
UNCLASSIFIED DECL IDAH030579 IDA/HQ-85-30579 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.2
2.0
1.8

Work Package

Identification: R

Name: Correlations between the ANSI/MIL-STD 1815A Ada Informal Definition and the Ada FD.

Purpose: To correlate the existing informal and the resulting formal definitions of Ada.

Contents:

It has been suggested that eventually the ISO will adopt an Ada standard which consists of two parts: an informal, and a formal one -- much the same way as the CCITT has both an informal and a formal definition of the CHILL language.

Also, to avoid, i.e. to attempt to alleviate (as far as is possible for pairs of informal and formal definitions) discrepancies between these, a systematic attempt must be made to correlate them.

Finally such a correlation also serves to make the Ada FD more accesible.

The work consists of producing two pairs of annotated documents, both electronically maintainable: one, derived from the Ada FD, which correlates its formulae to the ANSI /MIL STD 1815A document, and, another, derived from this latter document, which correlates its sentences and paragraphs to the formulae of the Ada FD. Both these documents may need further, generally explicative notes.

Requisites: pre: WPs I-J-K-L
post: None (I-J-K-L (!))

Man Months: 4

Deliverables: Report 31: An ANSI/MIL-STD 1815A Ada Reference Manual to Ada FD correlation.
Report 32: An Ada FD to ANSI/MIL-STD 1815A Reference Manual correlation.

This deliverable will be issued in two stages: ½31-½32 based on ½ stage WP I-J-K reports, and 31-32 based on final WP L reports.

Review: Reports ½31-½32-31-32.

Project Description

Work Package

Identification: S

Name: Feasibility Study: ACVC vs. Ada FD Validation

Purpose: To ascertain the extent to which the Ada FD may serve as the direct, or indirect basis for a validation of the ACVC test suite.

Contents:

It has been argued that the Ada FD should, or could, be used as the basis for a formal verification of the ACVC test suite of correct and incorrect Ada programs. The purpose of this work package is to study the feasibility of this thesis. Different approaches are conjectured:

- direct executability of the Ada FD
- (automatic, or interactively assisted) proof/disproof of properties of each individual ACVC program
- indirect executability via either the Karlsruhe EAG, or the NYU SETL descriptions, or both -- either of which have formally, or systematically been shown "equivalent" to the presently proposed Ada FD.

Requisites: pre: WPs I-J-K-L-Q
post: none

Man Months: 3

Deliverables: Report 33: Feasibility of ACVC validation with respect to the Ada FD

Review: No

Work Package

Identification: T

Name: ISO (ANSI, ECMA) Liaison

Purpose: To guarantee that the present project results in a FD, which

- reflects as much as possible of the current state of Ada as discussed within ISO, and
- may possibly influence Ada changes in the 1st 5 year Ada review by ISO, and
- will be ultimately acceptable by ISO as part of their subsequent Ada Standard.

Contents:

Travels to ISO Ada standardisation meetings, and correspondance with other organizations as determined from wp A.

It may be expected that the ISO liaison may lead to desire by ISO or other official institutions that the current project attempts to work out proposed changes to the ANSI/MIL-STD 1815A (January 1983). The present project has not included this in the ressource estimates, and does not intend to do so.

Requisites: pre: None
post: None

Man Months: 4

Deliverables: Unnumbered reports: travels, deliberations and status

Project Description

Work Package

Identification: U

Name: Management

Purpose: To coordinate internal work packages, external liaisons and reviews, partnership sub-projects, and CEC liaison.

Contents:

Establishment, monitoring and control of rolling plans and resources, budgets and finances.

Man Months: 26

Deliverables: Monthly and ½ year reports to CEC

Work Package

Identification: V

Name: External Reviews

Purpose: To guarantee quality and acceptance of resulting deliverables.

Contents:

International groups of Ada and FD experts will be established, consisting of experts in the relevant fields as well as representatives from relevant part of the Industry. Their members will be agreed upon by the CEC and the contractors. The groups will be referred to as the Ada FD advisory groups. These groups will regularly receive draft and proposed final reports of the various Ada FD, informal correlations, etc. The review process is then one of obtaining input on the form and content of these documents. This will insure that all achademic points of views are taken into consideration as well as the practical use of the results.

The CEC and the contractors will set up a review board to assist in evaluating the results of the projects, using as a major input, the comments from the international advisory groups. See also sect. 6.3.5.

Funding of these activities is not included in this project, except for the contractor part.

Requisites: pre: WPs B, D, E (see WP F), G, I-J-K-L, P, R
post: WP -- accordingly

Man Months: 4

Deliverables: Reports: draft, and final review reports.

Project Description

Work Package

Identification: X

Name: Educational Material

Purpose: To plan a set of tutorial courses on the use of Ada FD and implement one of them.

Contents:

In order to enlarge the user group and make the Ada FD accessible to people not familiar with formal definitions, tutorial type courses will be planned.

The work package consists of 3 parts:

1. Focusing on the user groups defined in wp B, course contents will be defined for each, emphasizing the needs of that particular group.

2. Implementation of one of these (typically 2 week) courses and

3. holding a trial course.

Requisites: pre: B-G-I-J-K-L
post: none

Man Months: 10

Deliverables: Report 34: Tutorial needs of specific user groups Course notes

Report 35: Course notes and Instructors manual

Review: Report 34

6.3 Management Issue

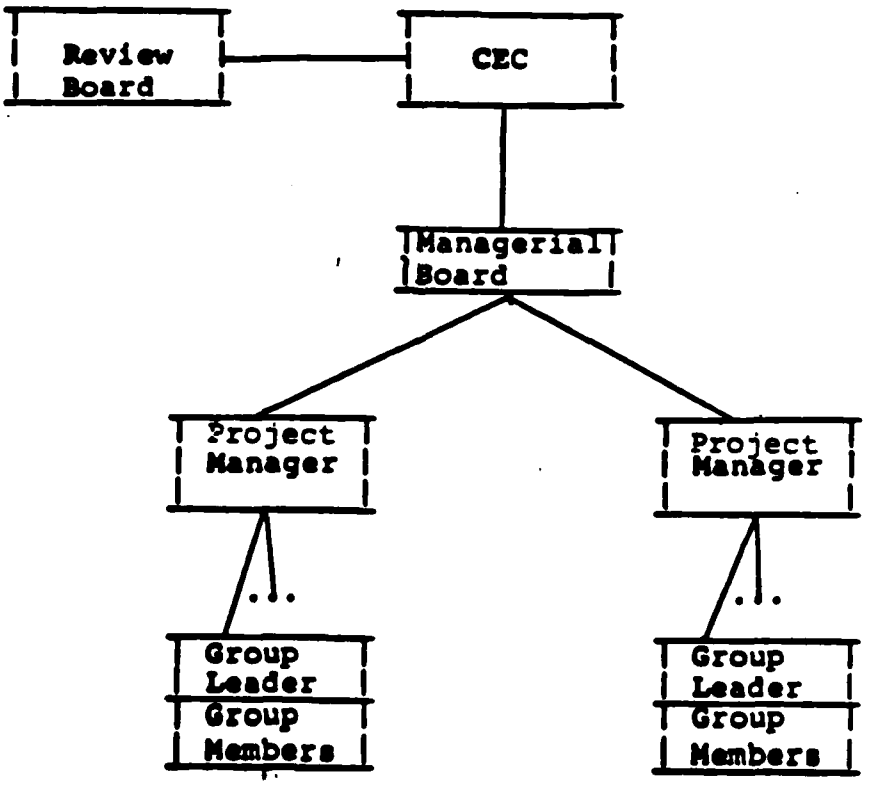
In this section two issues will be addressed.

- setup of the management organization

- work schedule and deliverable items list.

6.3.1 Project Organization

The project organization is defined in the following organization chart.



6.3.2 Managerial Board

The managerial board is responsible for all decisions affecting more than one partner, in particular:

- all contractual matters
- approval of all major technical decisions concerning requirements for components delivered
  - internally, by one Contractor to another
  - externally, by one Contractor to the Commission or other parties outside of the group of Contractors
- approval of significant changes in the development plan and all changes that affect the delivery of a contractual item to be submitted to the CEC
- monitoring the progress of work including quality control and quality assurance procedures.

The managerial board consists of one representative from each of the contractors. The representative must be able to represent his company in financial matters, and to negotiate with the Commission on behalf of his company. He will further endeavour to insure that his company satisfactorily performs the execution of tasks assigned to it.

To resolve major technical problems the managerial board may appoint fast working committees.

After having informed the others, each contractor shall have the right to replace its representative.

The Managerial Board shall be chaired by the Prime Contractors representatives.

It shall meet at least 3 times a year or, at every time when necessary at the request of one of the Contractors. Meetings shall be convened by the Chairman with at least seven days' prior notice with agenda.

A secretary shall be appointed by the members of the board. Minutes of the meetings of the Managerial Board shall be drafted by the secretary and transmitted to the Contractors without delay.

The Project Managers shall attend the meetings of the managerial board.

Decisions must be unanimous.

6.3.3 Project Managers

Each of the contractors appoints a Project Manager. The project manager appointed by the main contractor also acts as project coordinator.

Each project manager is responsible towards the managerial board for

- the coordination and scheduling of all project tasks assigned to his site
- the punctual delivery of any contractual item in project activities of his site
- definition of suitable programming and documentation standards to be followed in the project
- acceptance test procedures
- configuration, ordering, installation, and maintenance of any hardware required for the project
- reporting to the Managerial Board about the progress of the technical work
- presentation to the Commission and/or appointed technical experts
- maintaining contacts with the Ada related communities mentioned elsewhere.

In addition to the site manager each contractor will have a deputy project manager who will take over the responsibility of the project manager during any long-term absence of the project manager.

The project coordinator is additionally responsible for ensuring a continuous, consistent contact between managerial board and project managers.

Among the responsibilities of the project coordinator are:

- co-ordination af activity plan,
- co-ordination of documentation standards and all matters relevant to integration of the different projects sites,

- preparation and distribution of regular overall progress reports to the CEC,
- organization of presentations and review meetings etc.,
- maintaining the formal contact between the managerial board and the CEC,
- collection of the Contractors documents and statements of expenditures and forwarding thereof to the CEC.

6.3.4 Group Leader

Each group leader is accountable to his project manager for

- the execution of the tasks assigned to his group
- the correct performance and punctual delivery of these tasks in accordance with the specifications and schedules approved by the managerial board
- issuing regular progress reports for the technical work assigned to his group
- keeping the project manager informed about any important problem (technical as well as non-technical) that might arise in his group. In particular, the group leader must report immediately to the project manager any problem whose solution might involve a change in the resources allocated to the task
- co-ordination of all group technical activities including
  - adherence to standards
  - specification of work to be performed by group members
  - specification of important interfaces
  - quality control.

6.3.5 Review Board

The Review Board is a technical board which supports the CEC and the Contractors with technical advice during the project reviews and the project presentations. Its members are selected jointly by the CEC and the Contractors (ref wp V), and should represent all user interest.

6.3.6 Project Planning and Follow-up

Each contractor will set up procedures to plan and monitor his part of the project. The procedures will include:

- definition of internal milestones (contents and date); progress is measured solely on completion of milestones
- associating ressource estimates and allocating persons to each work package
- monthly reporting describing which results have been reached, the ressources consumed and the overall plans for the remaining part of the project (a rolling plan).

6.3.7 Project Reporting

For every calendar month there will be a Management Control Report to the Commission. The Report will be two A4 pages long and contain statements on:

- work package started in the last month
- work package completed in the last month
- work package delayed with reasons, and actions to be taken to correct
- work package scheduled for the subsequent month
- revised project plan if necessary.

There will be Financial Statements every six months.

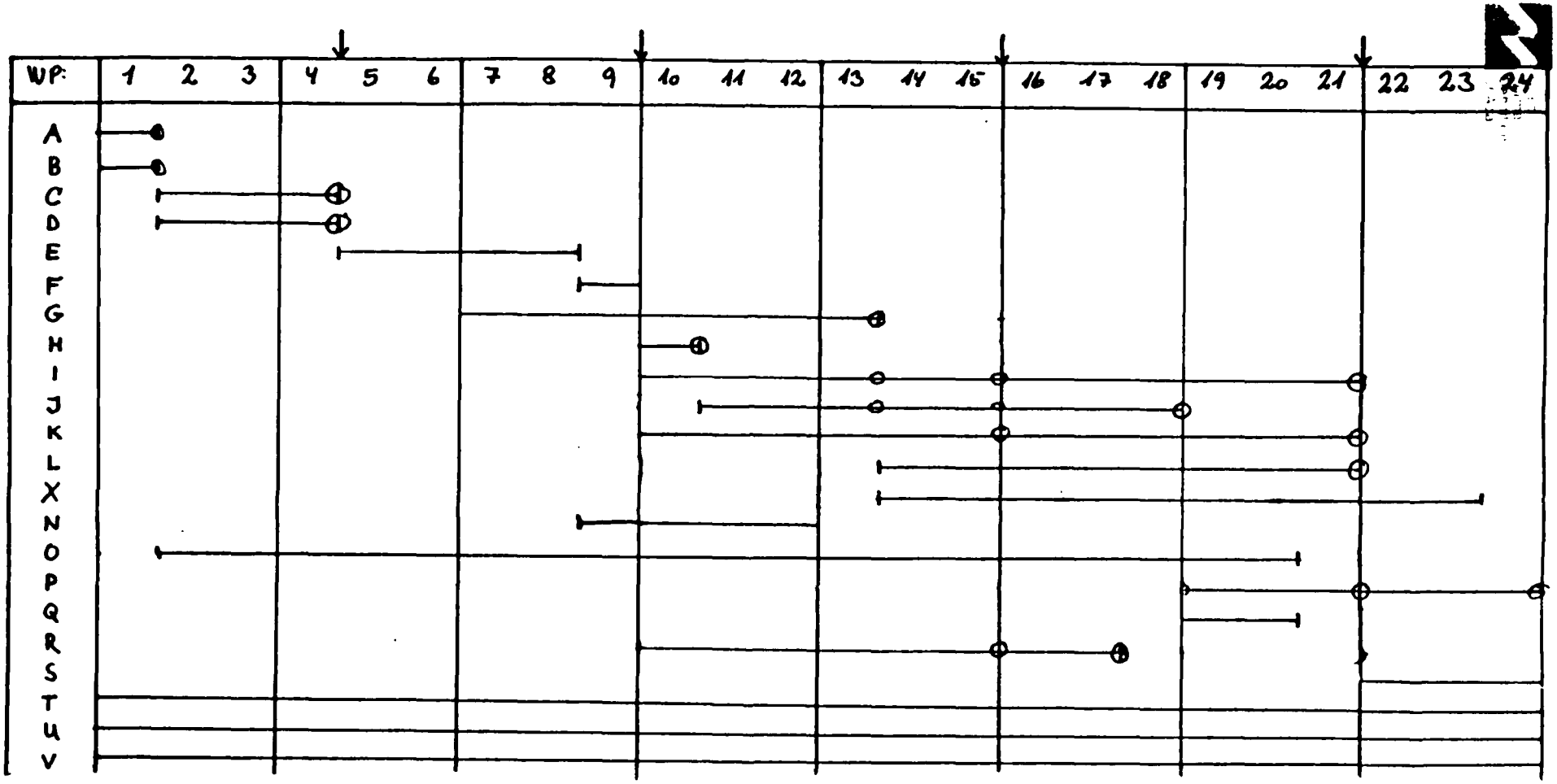
All project reports will be in English.

All deliverables to the Commission will be provided to all partners and subcontractors.

6.3.8 Work Schedule and Deliverable Items

This is the planned division of work which is subject to change during the project. Major Changes which influence the division of responsibilities between the partners, must be approved by CEC.

Time bar chart



O review points

→ project review

Relation to deliverables:

Please refer to wp descriptions.

DDC/CRAI

Manpower Resource Allocation: Month-by-Month/WP-by-WP -- First 12 Months

	1	2	3	4	5	6	7	8	9	10	11	12
A	1/0											
B	1/0											
C		1/0	1/1	1/1								
D			1/1	1/1								
E					2/1	2/1	2/1	2/1				
F									1/0			
G							0/1	0/1	0/1	0/1	0/1	0/1
H										1/0		
I										1/0	1/0	1/0
J											1/0	1/0
K										0/1	0/1	0/1
L												
N								0/1	1/1	0/1	0/1	
O		1/0	1/0	1/0	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
P												
Q												
X												
R										½/0	½/0	½/0
S												
T	1 person DDC one-sixth time, full 24 months											
U	1 person DDC 3/4 time, 1 person CRAI one-third time, full 24 months											
V	1 person DDC one-sixth time, full 24 months											

Sums:

	1	2	3	4	5	6	7	8	9	10	11	12
DDC	3X	3X	4X	4X	3X	3X	3X	3X	3X	3Y	3Y	3Y
CRAI	Z	Z	2Z	2Z	2Z	2Z	3Z	4Z	3Z	4Z	4Z	3Z
Total	3W	3W	6W	6W	5W	5W	6W	7W	6W	7U	7U	6U

X: 1/12, Y: 7/12, Z: 1/3, W: 5/12, U: 11/12

DDC/CRAI

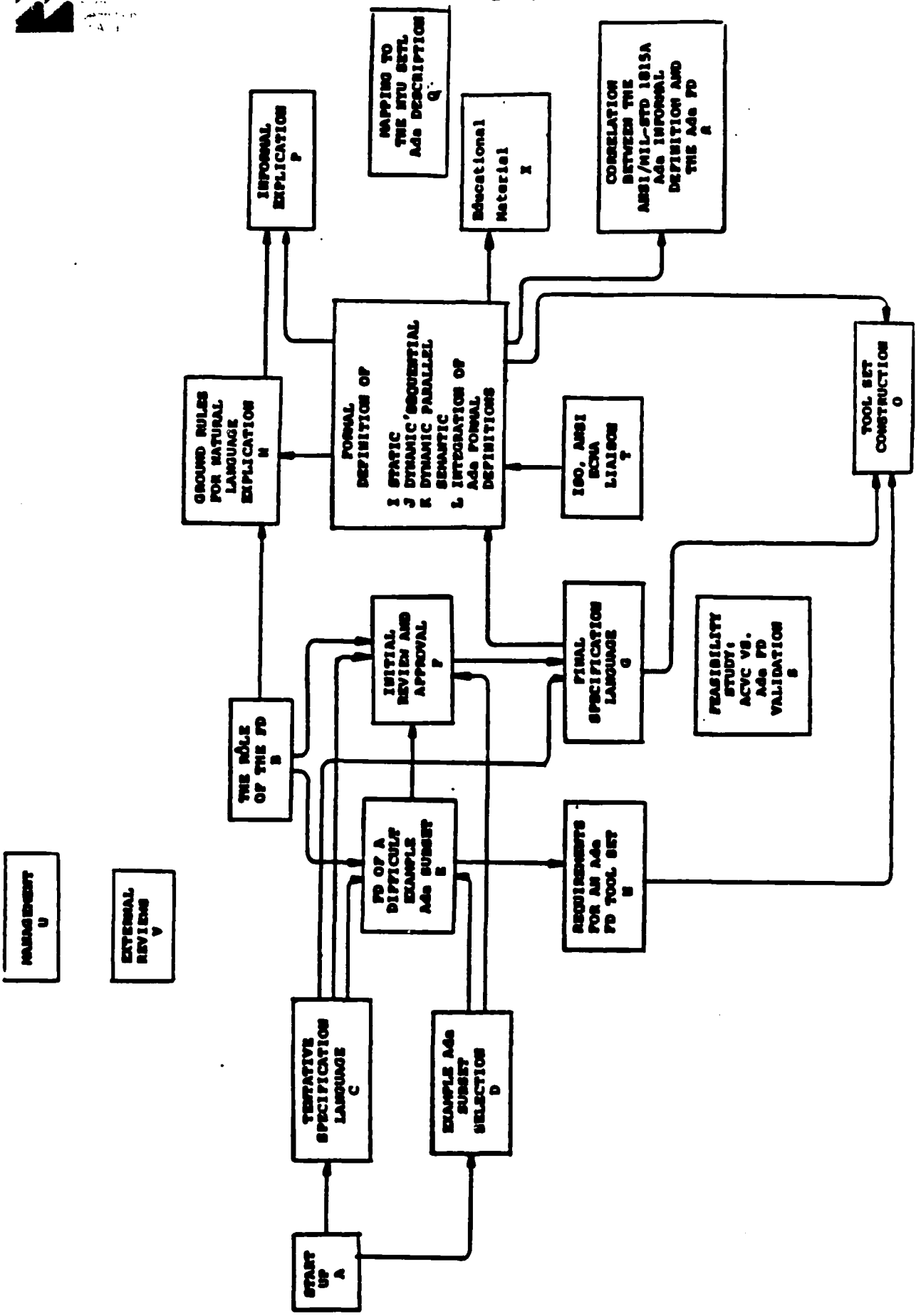
Manpower Resource Allocation: Month-by-Month/WP-by-WP -- Last 12 Months Summary

	13	14	15	16	17	18	19	20	21	22	23	24	Sum	WP
A													1/0	A
B													1/0	B
C													3/2	C
D													2/2	D
E													8/4	E
F													1/0	F
G	0/1												0/7	G
H													1/0	H
I	1/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0				12/0	I
J	1/0	1/0	1/0	1/0	1/0	1/0							8/0	J
K	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1				0/12	K
L		1/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0				8/0	L
N													1/4	N
O	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1					3/16	O
P							1/0	1/0	1/0	1/0	1/0	1/0	6/0	P
Q							0/1	0/1					0/2	Q
X		0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1		0/10	X
R	½/0	½/0	½/0	½/0	½/0								4/0	R
S										0/1	0/1	0/1	0/3	S
T	1 person DDC one-sixth time, full 24 months												4/0	T
U	1 person DDC 3/4 time, 1 person CRAI one-third time, full 24 months												18/8	U
V	1 person DDC one-sixth time, full 24 months												4/0	V

Sums:

DDC	3Y	4Y	4Y	4Y	4Y	4X	4X	4X	4X	2X	2X	2X	85 PM
CRAI	3Z	3Z	3Z	3Z	3Z	3Z	4Z	4Z	2Z	2Z	2Z	1Z	70 PM
Total	6U	7U	7U	7U	7U	7W	8W	8W	6W	4W	4W	3W	155 PM

X: 1/12, Y: 7/12, Z: 1/3, W: 5/12, U: 11/12


START UP A
TENTATIVE SPECIFICATION LANGUAGE C
EXAMPLE Ada SUBSET SELECTION D
THE ROLE OF THE FD B
FD OF A DIFFICULT EXAMPLE Ada SUBSET E
INITIAL REVIEW AND APPROVAL F
FINAL SPECIFICATION LANGUAGE G
REQUIREMENTS FOR AN Ada FD TOOL SET H
FEASIBILITY STUDY: ACVC VS. Ada FD VALIDATION S
GROUND RULES FOR NATURAL LANGUAGE EXPLICATION N
FORMAL DEFINITION OF
I STATIC
J DYNAMIC SEQUENTIAL
K DYNAMIC PARALLEL
SEMANTIC
L INTEGRATION OF Ada FORMAL DEFINITIONS
ISO, ANSI SCMA LIAISON T
TOOL SET CONSTRUCTION O
INFORMAL EXPLICATION P
MAPPING TO THE STU SETL Ada DESCRIPTION Q
Educational Material X
CORRELATION BETWEEN THE ANSI/MIL-STD 1815A Ada INFORMAL DEFINITION AND THE Ada FD R
MANAGEMENT U
EXTERNAL REVIEWS V

Project Description

WP	Name	Deliverable Items
A	Start up	Project and Review procedures, mailing lists
B	The Rôle of the RD (Formal Definition)	Report: The Rôle of the FD of Ada
C	Tentative Specification Language	Report: informal description of the specification languages and notations to be used.
D	Example Ada subset selection	Report: references to those parts of the Ada languages, by reference to the ANSI Ada informal description, which will be subject to a FD in WP E.
E	FD of a "Difficult" example Ada subset	Report: FD of "Difficult", Example Ada Subset
F	Initial Review and Approval	Report: review -- with positive/negative recommendations: whether to continue, or redo parts of WP E.
G	Final Specification Language	Report: informal description and formal definition of the individual and combined semantics of the chose specification languages and methods.
H	Ground rules for natural language explication	Report: Guidelines on how to extract and correlate informal descriptions of Ada from/to an Ada FD.
I	Formal Definition of Ada Static Semantics	Report: The Formal Definition of Ada Static Semantics.
J	Formal Definition of Ada Dynamic Sequential Semantics	Report: The Formal Definition of Ada Dynamic Sequential Semantics.
K	Formal Definition of the Ada Dynamic Parellel (ie Tasking) Semantics	Report: The Formal Definition of Ada Dynamic Tasking Semantics.
L	Integration of Ada Formal Definitions	Report: The Formal Definition of Ada

WP	System Name	Deliverable Items
N	Requirements for an Ada FD Tool set	Report: Requirements for a Portable, APSE-based Ada FD tool set.
O	Tool set Construction	Report: Ada FD tool set: architecture and design, users manual, installation manual, primer, etc. Software: Portable, APSE-based Ada FD Tool set.
P	Informal Explication	Report: An Informal Explication of the Ada FD.
Q	Feasibility Study: Mappings to the NYU SETL Ada Interpreter	Report: Feasibility of a Mapping from the Ada FD to the NYU SETL Ada Interpreter.
R	Correlations between the ANSI/MIL-STD 1815A Ada Informal Definition and the Ada FD.	2 Reports: The Ada Reference Manual to Ada FD Correlation, and: The Ada FD to Ada Reference Manual Correlation
S	Feasibility Study: ACVC vs. FD Validation	Report: Feasible Ada FD related models for ACVC test suite validations.
T	ISO (ANSI, ECMA) Liaison	Travel and Status reports.
U	Management	
V	External Reviews	Reports: draft, and final review reports.
X	Educational Material	Report: Tutorial needs of specific user groups.

All items with exception of software developed under workpackage O and management reports are public.

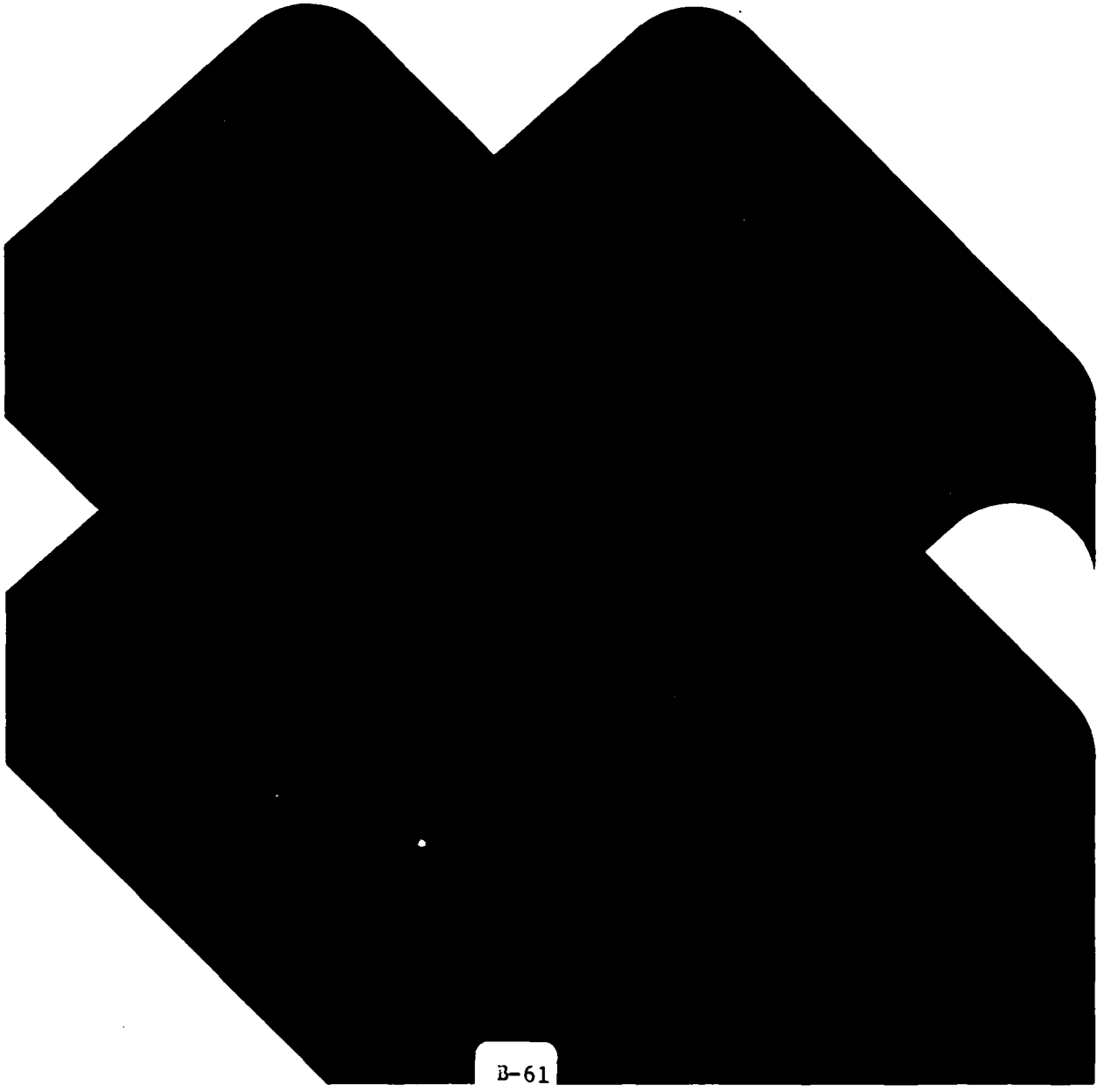
The Draft Formal Definition of Ada®

Commission of the European Communities: Multi-Annual Programme

Towards a SMoLCS Based Abstract Operational Model for Ada

E. Astesiano, F. Mazzanti, G. Reggio

20 August , 1985



ABSTRACT

This document is a deliverable related to the activity of work packages C and D.

In this report the motivations of the choice of an underlying model for Ada and its main features are outlined.

TABLE OF CONTENTS

1.0 INTRODUCTION

We agree with the opinion strongly put forward by E.K.Blum in [Blum 84] that any reasonable semantics of Ada should rely on an underlying semantic model, and that we can be misguided if we just look for a syntax directed semantics without a preliminary study of such a model. This is particularly true of Ada which follows a sequential declarative/imperative style even when introducing concurrent features (e.g. the syntactic construct for the task creation has the form of a declaration), so that there is no easy relationship between the text of a program and the states of its executions.
Nevertheless, since the methodological importance and the acceptance of the syntax directed approach is out of question, we think that an effort should be made for giving a syntax directed semantics relying on a clearly defined and understood underlying model.

In this report we will outline the motivations of the choice of a model and its main features. In another report [Astesiano et al. 85 b] it is shown, on some sample languages with Ada-like features, how to connect an underlying model to a syntax directed approach.
We start from the assumption that the behaviour of an Ada program is represented by a concurrent (flagged) transition system specified in the SMoLCS style (see [Astesiano 84] for an informal introduction and [Astesiano et al. 85 a] for foundations) .
We want to investigate the overall structure of this specification, i.e. the possibility of defining it in an inductive way as well as the overall structure and meaning of the needed information, structures and actions.
In defining this operational model we try to remain as much abstract as possible, modelling explicitly only the semantically "non-hiddable" features of the language.

In the project, the semantics of Ada will be given in two parts: a first one specifying the static semantics of the language, and a second one specifying its dynamic (sequential and concurrent) aspects.
As this model will be used as a reference in the description of the dynamic semantics of Ada, it needs not to reflect all the properties already stated in the static semantics part.
This approach is rather different from the standard one adopted in giving denotational semantics, in which often static and dynamic semantics are specified together (e.g. modelling type checking as if it occurred dynamically); indeed we assume that our programs are always correct from the point of view of static semantics (e.g. scoping and type checking).
Being our model more dynamics-oriented and less statics-oriented, it seems reasonable to make some assumptions about the syntax of the program, which may simplify the overall description

of the model.

For example, a useful assumption is that all the identifiers (introduced by declarations) appearing in different places in the source text are different. This simplification allows to associate to each name, at each point of the text, a unique meaning; with this approach indeed, problems of hiding or overloading are supposed to be resolved almost completely in a previous step consisting in a simple translation and checking of the source Ada text.

2.0 LOOKING FOR AN INDUCTIVE STRUCTURE

A basic idea of the SMoLCS methodology is that a concurrent system is modelled as a labelled transition system whose states consist of the states of the component processes (subsystems), plus some global information. These states are usually represented as couples

$$\langle s_1 \mid s_2 \mid s_3 \mid \ldots \mid s_n , \text{inf} \rangle$$

where $s_1 \mid s_2 \mid s_3 \mid \ldots \mid s_n$ is a multiset of states of the component processes.
The transitions of a state of a concurrent system are inferred from the transitions of the component processes by means of formulas of the form:

$$\text{cond} \wedge s_1 \xrightarrow{f1} s_1' \wedge s_2 \xrightarrow{f2} s_2' \wedge s_n \xrightarrow{fn} s_n' \supset s \xrightarrow{f} s' .$$

Given a basic transition system, specifying the structure of the component processes, and some parameters related to synchronization, parallel composition and monitoring, we can produce the final transition system in a canonical way using the SMoLCS methodology.
If the component processes are themselves concurrent systems, they can be specified in the same way. A SMoLCS specification of a system may be inductive, as it is typical of the SOS approach [Plotkin 81].

It is not evident how an Ada program might be mapped into one of these inductive or hierarchical structures in a rather natural way. Several alternatives are discussed, together with their advantages and weak points.

First of all we must give an intuitive meaning to the hierarchical concept of "subsystem". In general it is useful to associate a subsystem with the execution of some syntactic construct of the language. For example, if we want to put in evidence the concurrent structure of a program, we might represent by a subsystem the behaviour of a task; otherwise we might represent by a subsystem the behaviour of a block, or task, or procedure, putting in evidence the nested structure of all constructs.
Then we must define which is the relation defining when a subsystem is a component of another subsystem, and what is the information modelled in each subsystem

We begin with a discussion of two rather interesting alternatives, based respectively on the mastership dependence between masters and tasks and on the environment/store structure.

2.1 *A dependence driven dynamic structure*

In this section we suppose that an Ada program is composed exclusively by tasks (types) (i.e. no

subprograms nor blocks are used); this allows us to give a subsystem a simple meaning (task execution). This restriction might be removed easily but now it allows us to put in evidence with the least effort and without any loss of generality the problems related to the hierarchical decomposition of an Ada program.

In this model the direct inclusion relation between subsystems explicitly represents the direct dependence relation between tasks, i.e.

if sub1 is the subsystem corresponding to the task T_1, and
s is the subsystem corresponding to the task T,
then sub1 is a subsystem of s <==> T_1 is a direct dependent of T.

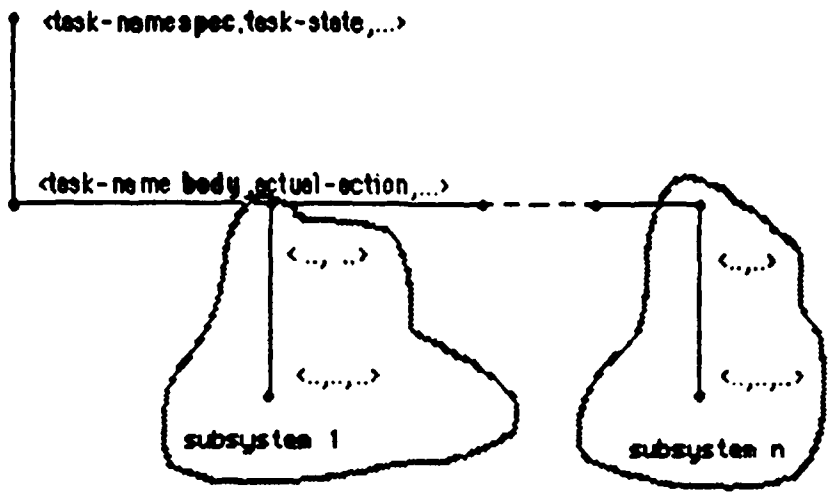
When a new task X is created, a new subsystem is created within the subsystem corresponding to the task "master" of X. In this way, the structure of the system explicitly models the dependence structures within a program. We can observe that in this way the subsystems corresponding to all the tasks which are direct or indirect dependents of a task T, are included (at various levels) in the subsystem corresponding to T.
As a consequence, all the interactions between tasks based on this dependence relation are modelled in a very natural and simple way.

For example the effect of an abort statement on a task is completely defined as a transformation of the corresponding subsystem (this subsystem and all its component subsystems become "abnormal" or completed), allowing a direct representation of what is specified in the manual.

If we suppose that a subsystem is represented by the following scheme :

< task-name, task-state, actual-action , subsystem 1 | ... | subsystem n>,

or using a graphic notation (following the style illustrated in [Bjorner et al. 80]):



(note that these graphs can be formally defined)

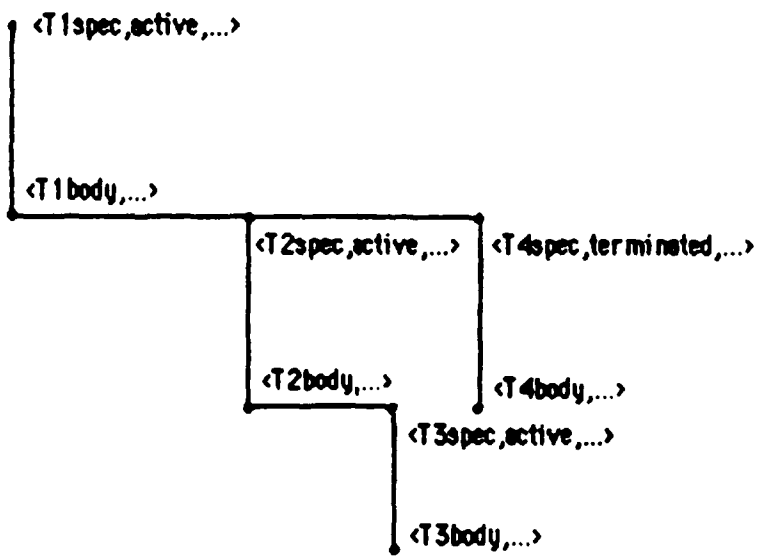
then the effects of an abort statement on the task T1 can by illustrated, for example, by the following transformation of the corresponding subsystem:

<T1, active , ... ,<T2, active , delay(...), <T3, active ,if ....>>| < T4,terminated,...>

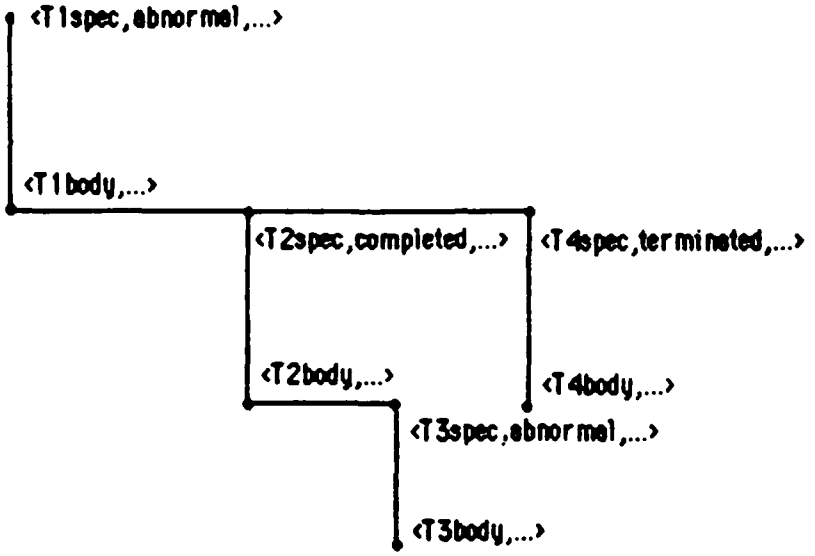
becomes

<T1,abnormal,...,<T2,completed, delay(...),< T3,abnormal,if ...,>>| < T4,terminated,....>>

or using the graphic notation:



becomes:



Analogously task <u>termination</u> is easily modelled as occurring when the corresponding subsystem is "completed" and all its component subsystems (if any) are "terminateable" (a subsystem is said "terminateable" when the corresponding task is terminated or suspended on a select statement with an open terminate alternative and all its subsystems are "terminateable"), following strictly the manual description of this event.

Apart from allowing a natural representation of these synchronized actions, this model presents some disadvantages in describing other kinds of interactions, not related to the definition of the dependence relation between tasks.

As a consequence of the Ada definition of dependence, it may happen that when a task is created by the evaluation of an allocator, its master is not the task which has created it but some other task at a higher level, more precisely the task which has elaborated the corresponding access type declaration. In this model the request for task creation has to propagate upwards and the subsystem corresponding to the creating task has to interact with some higher subsystem (by synchronization or by reading its information) in order to deal with the activation of the created tasks .

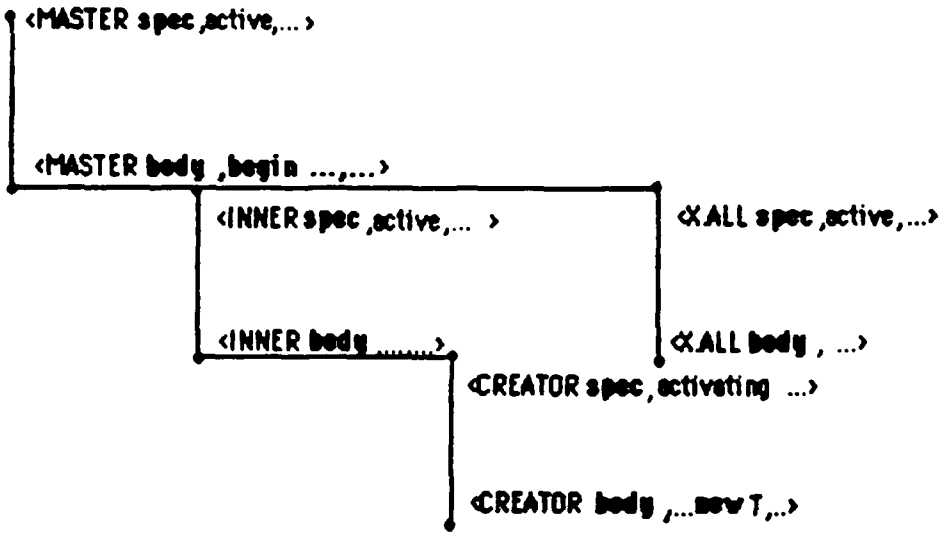
This is in particular illustrated by the following simple example:

```
task MASTER;
task body MASTER is
   task type T is ... end T;
   type RT is access T;     -- the task "MASTER" declares the access type
                            -- designating T
   task body T is ... end T;
   task INNER;                    -- this is a nested task within the task "MASTER"
   task body INNER is
      task CREATOR ;              -- this is a nested task within the task "INNER"
      task body CREATOR is
         X : RT := new T;         -- the task "creator" activates an instance of T
      begin                       --   (which is a dependent of "MASTER")
         ...
      end CREATOR;
   begin
      ...
   end INNER
begin
  ...
end MASTER;
```

the situation during the evaluation of the allocator new T is illustrated below:

<MASTER,active,begin...,<INNER,active,...,<CREATOR,activating,...newT>>,<X.ALL,active,...>>

or using the alternative graphic representation:



In this case the subsystem corresponding to the task "XALL" is created at a higher level than the subsystem corresponding to the task "CREATOR" (waiting for the completion of the activation of XALL).

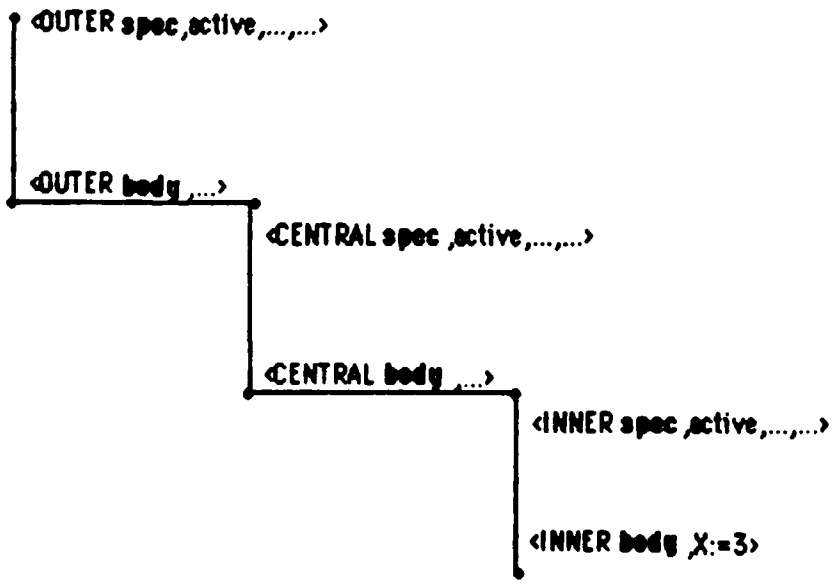
This hierarchical structuring should allow us to split the description of the environment and the store at different levels in the model following the principle of "information hiding". Reasonably, we may think to associate to each subsystem the local environment and store defined by the elaboration of the declarative part of the corresponding task. References to local definitions and objects (i.e. defined in the declarative part of the corresponding task) are described rather naturally; references to non-local definitions and objects instead are represented in an unnatural heavy way. In fact the dependence relation between tasks does not correspond to the natural structure of the environment and so it may happen that references to non-local entities (here and in the following we mean by entity an Ada entity, and for object an Ada object /LRM 3.1(1), 3.2(1)/) have to propagate upward through subsystems which should be not visible nor accessible to the task (this is not a nice property). This situation is illustrated in the following example:

```
task OUTER;
task body OUTER is
     X : INTEGER;              -- in a declarative part are declared  some entities
     task type T;              --  and a task type T
     task body T is
      begin
         X : = 3;              -- task instances  use some previously declared objects
     end T;
     task CENTRAL;             -- an other task is declared in the same declarative part
     task body CENTRAL is
           Y : INTEGER;        -- entities and objects  may be declared locally
          INNER : T;           -- but are not  not visible from an instance of the task type T
      begin  ...
     end CENTRAL;
 begin   ...
end OUTER;
```

the execution of X:=3 by the task INNER is illustrated below:

<OUTER,active,..., < CENTRAL,active, ...,< INNER,active, X:=3 ,...>>>

or using the alternative graphic representation:



Though INNER is a subsystem of CENTRAL, its entity accesses should propagate at least to the OUTER subsystem (skipping the CENTRAL subsystem).

These references in fact should in general be propagated from the subsystem corresponding to the task which has created a new task NT, to at least the subsystem corresponding to the task which has elaborated the declaration of the type of NT .

One of the important properties which make reasonable a hierarchical structure is that from the outside of a subsystem it is not required direct visibility of the component subsystems. This property is verified by this model only partly; indeed, though the local environment and store of a subsystem can be used only from inside of the subsystem, other information about the subsystem itself must in general be visible from higher levels. A task of a given type T may be in general accessible to other tasks whose type is declared at the same level of T.

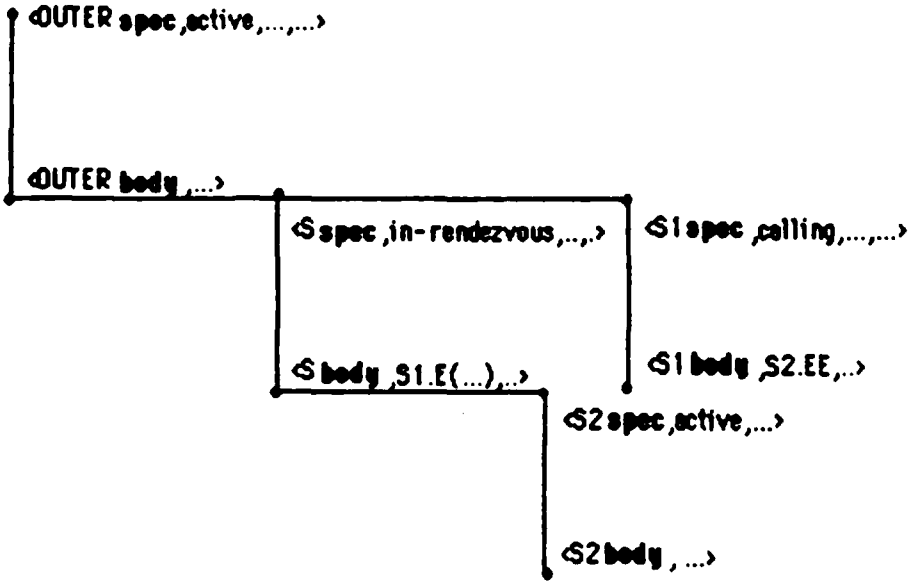
In this model it is not difficult to build an example in which a subsystem S1 might interact (having visibility of it) with another more nested subsystem S2. This situation may occur when tasks are passed as parameters in entry calls as illustrated in the following example.

```
task OUTER
task body OUTER is
   task type T is entry EE; end T;        -- a task type T is declared
   task body T is ... end T;
   task S1 is entry E (S2 : T); end S1;   -- a task object is created,
   task body S1 is
      begin                               -- willing to receive, as an entry parameter
         accept E (S2 : T) do S2.EE end accept; -- a task object whose type is T
    end S1;
   task S ;                          -- in the same declarative part of the other tasks
   task body S is                    -- an other task object S is created
    S2 : T ;                         -- and creates a task object (of type T)
    begin                            -- (the new instance is a dependent of S itself)
      S1.E (S2);                     -- the new instance is then passed to S1
    end S;
begin ...
end OUTER;
```

the situation during the call of S2 .EE is illustrated below:

‹OUTER,active,... ,‹S1,calling, S2.EE,›|‹S,in-rendezvous, S1.E(...),‹S2,active, ...›››

or using the alternative graphic representation:



S1, which is at a higher level than S2, issues an entry call to S2, which may be not willing to accept it.

The conclusion is that only part of the information about a task could be modelled within a subsystem (i.e. information about local environment and store), and that other structures are still needed in order to maintain the information about rendezvous (queues, state of entries...), task attribute's values ('CALLABLE, 'TERMINATED, 'COUNT), state of the task (activated, in activation,...).
This information about a task, which cannot be directly modelled within the corresponding subsystem, could either be defined as global or split at different levels in the dependence driven structure.

But even assuming as global the needed information, still we have the problem that modelling rendezvous requires some kind of synchronized action between subsystems in different branches of the structure , which are at least " unpleasant " to represent. A possible alternative might be not to model this kind of actions as synchronous actions, for example modelling rendezvous by means of message exchanges, using shared memory, between processes

(indeed in this case an explicit abstract representation of task synchronization would be lost).

The situation does not change if we release the initial hypothesis that a program is composed exclusively by tasks (types) and either we associate a subsystem to the execution of each master (task, block, subprogram) of the program, or we associate a subsystem to task executions only.

Note: The problems illustrated previously are put in evidence by the border-line case of a function returning a task which was a dependent of the function body.
In this case we have that the subsystem corresponding to the function body no longer exists (because the function is terminated) though the task corresponding to one of its components is still accessible, for example it can be checked for termination (evaluating its 'TERMINATED attribute, or trying to start a rendezvous) (really it is terminated!) (information about subsystems which might no more exist should be kept somewhere).

2.2 *A scope driven structure of Ada programs*

Also in this section we consider first the case in which an Ada program is composed exclusively by tasks (whose behaviours are modelled by subsystems).

In this model, if a task T declares several task types, the subsystems contained in the subsystem associated to T are the subsystems associated to all the instances (tasks declared as objects, created by allocators or as subcomponent of other objects) of these task types; i.e

if sub1 is the subsystem corresponding to the task T_1 ,and
s is the subsystem corresponding to the task T,
then sub1 is a subsystem of s <==> type of T_1 is declared in the declarative part of T.

In this way the dynamic structure of a program directly reflects the actual structure of the environment (and store). This approach has been adopted in other operational models for sequential/parallel languages (see [Berry 71] [Johnston 71]).

Some advantages of this approach are its similarity to other standard operational models for sequential languages, and the fact that in this way references to local and non-local definitions and objects are represented in the most natural and simple way.

On the other side we should not need to split the information about a task at different levels of

our structure, because almost all the visible properties of a task could be recorded at the same level of the corresponding subsystem (scope rules assure us that a component of a subsystem is not visible from outside of the subsystem).

As in the previous case the effect of a task creation should propagate upward, in order to require the creation of a new subsystem, at the level of the declaration of the corresponding type (and similarly it happens for subprogram calls).

But a major problem with this approach is posed by the representation of all the "dependence driven" synchronized actions.

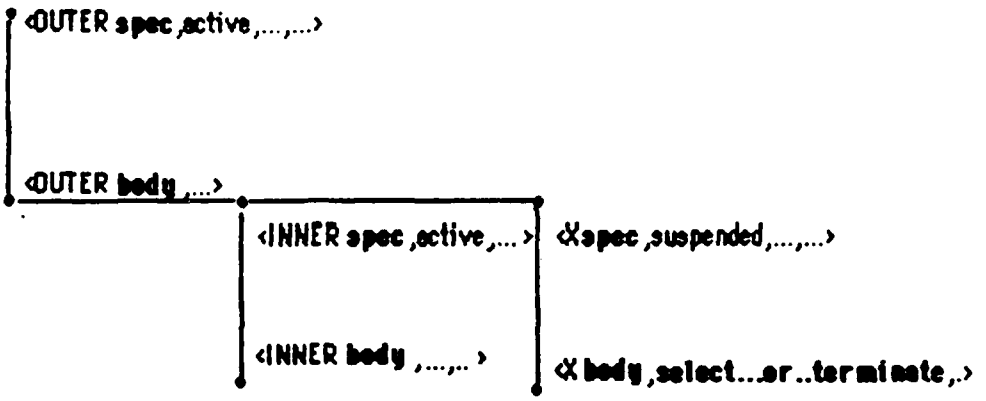
For example, starting with termination problems, a task suspended on a select statement with an open terminate alternative should look at the state of its master (is it completed?) in order to decide whether or not to terminate, even if its master is not visible in its environment (it is an usual case in Ada). This situation, requiring otherwise an access to a deeper subsystem, destroys our hope of keeping all the information about a task local to the corresponding subsystem. The following is a simple example of that:

```
task OUTER;
task body OUTER is
   task type T is  entry E;  end T;        -- a task type T is declared
   task body T is
    begin
       select  accept E; or  terminate;  -- whose instances may execute a selective
                                         -- wait
      end select.                        -- (they can terminate if their master is
                                         -- completed)
   end T;
   task INNER;                                -- a task object INNER is created
   task body INNER is
      X : T;                                  -- which creates a new instance X of the
                                              -- task type T
    begin  ...                                -- ( X is a dependent of INNER)
   end INNER;
begin ...
end OUTER;
```

the situation during the execution of the selective wait is illustrated below:

<OUTER,active,... ,<INNER,active,...,>|<X,suspended, **select ..or ..terminate** ,...>>

or using the alternative graphic representation:



Obviously X does not have any visibility of INNER which might be nested even more deeply.

Moreover the resulting synchronized action (of the termination of a master and all its dependent tasks) should involve several subsystems in general spread across the structure in a completely arbitrary way.

Exactly the same problem arises about the abort statement: the tasks which should become abnormal or completed are spread, without any constraint, across the whole structure.

It follows necessarily that the actions of this kind should be treated as top level synchronous actions (a possible alternative might be not to model this kind of actions as synchronous actions, for example modelling conditional termination by means of message exchanges between processes; the disadvantages of this approach have already been mentioned).

We can observe that even synchronous actions related to rendezvous may involve subsystems located at different levels of the structure, but in this case the situation is not very different from the previous model, and it is a direct consequence of the language features (allowing synchronized actions between tasks at different levels in the environment).

Nevertheless we can observe that in this case, as a consequence of the natural structure of Ada programs, entry calls can never occur from the outside of a subsystem towards an inner subsystem, but always from the inside towards outside (a task is always within the scope of the type of the task that it is calling).

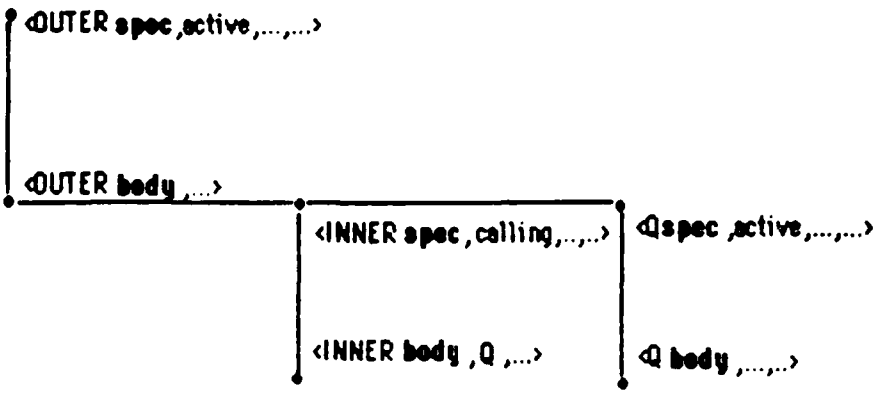
In the case of full Ada programs (i.e. releasing the previous restriction), in order to preserve the advantages of a scope driven structure, we must represent subprogram executions like task executions (associating a subsystem to each subprogram activation, at the level in which the subprogram was declared). But in this way we loose an explicit model of concurrency, treating in a uniform way sequential and concurrent constructs and mixing these conceptually different aspects.
The next example illustrates this issue:

```
task OUTER;
task body OUTER is
    procedure Q is ... end Q;
    task INNER ;
    task body INNER is
     begin
        Q;
    end INNER;
begin ...
end OUTER;
```

the situation during the execution of Q is illustrated below:

< OUTER,active,... , < INNER,calling,Q,,>,< Q,active, ...> >

or using the alternative graphic representation:



We can observe the similarity in the representation of Q and INNER (a procedure and a task).

2.3 *Motivations for a linearized model*

Looking at the problems which arise when we try to model inductively and hierachically an Ada program, we have to ask whether these problems are related to the model (we have not found the right structures in order to describe properly an Ada program) or to the language (does it really exist a semantic inductive structure corresponding to the inner structure of the language?).

We can in fact observe that an Ada program execution is driven by means of many different structures (dependences, activations, scoping) without a main one. Whichever relation we choose, in order to define hierarchically our model, we should deal in every case with the introduction of some kind of global information, and/or with an heavy representation of some kind of synchronized actions.

So we think that perhaps the best solution is not hierarchical, but a single level one.
In other words we directly model a program as a set of concurrent processes (all at the same level). Apparently this choice may seem lacking abstraction, but in the end it is the choice that models in the closest way what is expressed in the manual :"Tasks are entities whose execution proceed in parallel in the following sense. Each task can be considered to be executed by a logical processor of its own. ...". also the rendezvous is described without any reference to a structure. And perhaps it is not a case that other semantic models for Ada (e.g. [Bjorner et al. 80] [Dewar et al. 83]) have adopted almost the same approach. Environment, memory, dependences, and other information might be modelled as global and shared.

In conclusion this solution seems to avoid all the major disadvantages of the previous cases,

though it loses some of their advantages.

However note that modelling all the tasks at the same level, does not mean that we have to abandon an appropriate structuring global information (e.g. scope-driven for environment and store).

3.0 GLOBAL INFORMATION

In this section we illustrate the requirements (and their motivations) about the structure of the global information in the linearized tasking model.

In the SMoLCS approach the local information of a subsystem is not directly visible from other subsystems at the same level. This implies that whenever a subsystem needs some local information about an other subsystem, it may obtain that either looking in some global part (global information) or by means of a synchronized action with the other subsystem during which the needed information is received.

Unless we want to model non-local memory and environment accesses by means of synchronized actions (with an unpleasant confusion in the representation of concurrent and sequential aspects of the language), we have that memory and environment should surely be shared (global) information, as well as other information, like dependence relationships, task states and entry states, which is in general needed by more than one task.

On the converse, the information used only by a single task might be represented as local information of the task itself (for example the set of the names of the tasks to be activated after the elaboration of a declarative part).

An overall meta-requirement over all the structures within the FD is that they should be kept as abstract as possible, i.e. they should make explicit only the semantically relevant properties for which they are introduced, without the addition of implementative details. Moreover whenever possible, we would like to follow a standard way in defining these structure (for example in the case of the environment).

3.1 Environment

It should be clear from what said in the introduction, that the environment represented in this operational model need not to reflect all the properties already stated from the static semantics step. This means that the informal meaning of the "environment" is not a structure used to represent the set of associations (between identifiers and entities) which are visible in a certain instant, but rather a structure used to maintain the needed associations in order to model the execution of an Ada program (these associations may be more than the really visible ones, provided that the resulting effect does not change).

We think that the environment should include only those associations, between identifiers and Ada entities, introduced by the elaboration of (explicit or implicit) declarations. Objects (Left-Values), which are not directly named by declarations (e.g. subcomponents) should be obtained by the application of basic operations (indexing, selection of components) over composite objects.

In our model we would like to follow the standard style, where the environment is static outside a declarative part, i.e. where associations cannot change as a side effect of statements.

It is not so obvious that this is the most correct model for the environment in Ada, because of the presence of dynamic objects (objects whose internal structure may change as effect of statements) as is illustrated in the following example:

```
          -- V is an unconstrained variable whose type is R
<<A>>     V := (3,('a','b','c'));          type R (N : integer:=1) is
          ...                              record
          V.TEXT(5) := ('d');               TEXT : string(N);
           -- it raises CONSTRAINT ERROR   end record;
          ...
<<B>>     V := (5,('d','e','f','g','h'));
      ...
      V.TEXT(5) := ('d');
          -- it works successfully
```

It is not clear that the denotation of V should be considered the same (at <<A>> and <<B>>), provided that the application of a selection and a subsequent indexing operation on V produce different results (as Left-Values).
The problem arises essentially if we want to consider the information about the structure of V as part of the denotation of V.

Even assuming that dynamic objects do not create any problem, we are not allowed to consider all the associations of the environment as constant ones. In fact, because of the two step introduction of entities (program units, task types, deferred constants, incomplete types, recursive types) we must explicitly deal with updates of already introduced denotations in the environment.

This "limited kind of dynamicity" however has a deeper influence, on the structure of the environment, than one would expect; for example it does not allow to pass to each construct a copy of its own environment, avoiding in this way the existence of an explicitly shared environment (as it usually happens in the standard sequential cases). Problems, in this case, result out of the combination of the mixing of concurrency and of the two step introduction of entities (as illustrated in the following example).

```
 ...                          -- within a declarative part
task type T;                  -- an incomplete task type specification is introduced
 ...
package PACK is end PACK;
package body PACK is
     task INNER;
     task body INNER is       -- a new task INNER (using that specification as a
        type A is access T;   -- complete specification ) is then created
        REF : A;
      begin
        ...
        REF := new T;         -- INNER will activate a new instance of T
     end T;
begin
   null;                      -- the task INNER is now activated (during the
end PACK;                     -- elaboration of the body of the package PACK)
 ...
task body T is .... end T;    -- then the initial specification of T is completed
 ...
```

In this example it happens that a task INNER, which has visibility of an incomplete specification (T), proceeds in parallel with the elaboration of the rest of the declarative part. When the task INNER activates the instance REF.all of T, if the body of T has not been already elaborated then a PROGRAM ERROR exception should be raised, otherwise the task activation proceeds successfully. This is an evident example that, when the task T is defined (or even activated) (in our case INNER), it cannot receive a copy of the existing environment, because this environment may still be updated.

In general, name resolution (see the Overview) is not sufficient to uniquely identify the actual entity denoted from an identifier at a certain point of the execution. This is a direct consequence of the existence of recursive subprograms, of subprograms shared among tasks and of task

types (each one of the three cases would be enough). So we need some other kind of dynamic information in order to identify, at a given point of execution, the correct entity associated to an identifier (we shall call "environment-selector" this information). We can observe that for sequential languages, in which we are able to pass to each construct its proper environment, the structure of the environment itself can be a simple mapping from identifiers to denotations. In our case the environment is unique and shared; consequently it is more complex, for example it may have the form of a mapping:

(local-environment-selector x identifier) --> denotation.

We can observe how, in our case, the local-environment-selector plays the same role of the environment for sequential languages (as value passed from a construct to another modelling in an abstract way the present stack). During the elaboration of each construct, the present local-environment-selector should indeed record the position, in the "cactus stack like" structure of the global environment, corresponding to the present environment.

We do not discuss details about the structure of denotations in the environment, because it this is still matter of more detailed modelling.

3.2 *Memory Structure*

We recall that even in this case we should not give a particular implementation with a structure as "cactus stack" or "heap" but rather we should try to specify in an abstract way the requirements over the structure of the storage.

In our model we suppose that the memory is a unique, global and shared structure, containing all the associations between objects and their values.

In a different model of memory we might represent as global only the part of the memory explicitly shared between several tasks, still representing as local information of a subsystem the local part of the memory of a sequential task. We do not like this solution because in Ada it is not easy to distinguish between the local and the shared part of the memory of a task (which remains local until some inner tasks are activated).

The memory description, as it appears from the manual, it is not very abstract because often it refers to implementation dependent aspects of the language.

Moreover a really complete definition of Ada should not avoid to describe (at least some of) these implementation dependent features of Ada (for example the 'SIZE attributes or the STORAGE_ERROR exception) though the description of these features should be done without any loss of abstraction, for example by means of a parameterized specification. All these implementation dependent aspects might be introduced in fact as parameterized aspects within a unique store specification (as ADT).

As for the overall structure of this storage model, it should be seen essentially as a mapping from L_Values (Objects) to R_Values (Values)(as in the standard case). As in Ada objects and values can be composite, we can have that both R_values and R_Values may be complex (for example in the case of array); in this case we should be able to get the L_Value of a component from the L_Value of the whole object (and the same for the Values).
The correct correspondence between L_Values and R_Values might be stated by formulas within the storage specification, stating for example that the n^{th} component of a R_Values (corresponding to an array value) of a L_Value L should be equal to the R_Value of the n^{th} component of L; i.e. if A is a Left-Value corresponding to an array, R the "Rigth-Value" function (which given a L_Value and a memory state returns the corresponding R_Value) and I a Rigth-Value (R (A ,m)) (I) = R (A (I),m) .

This issue has been treated in a more complete way in a separate report [Reggio 85].
In an even more abstract model we could avoid to introduce explicitly L_Values, directly representing denotations as structured "complex" R_Values in the environment, treating in an axiomatic way problems about renaming and subcomponents. This last solution, even if feasible, seems too much abstract with respect to the Ada manual (in which objects are mentioned explicitly).

3.3 *Other Information*

As already said the information about a task needed by more than one task, which cannot be obtained with a synchronized action, should be represented as global.

The information about task dependences is just an important example. It might be represented abstractly as a relation between masters. This information should be global in order to model in a simple way statement like abort T, or other dependence driven (synchronized) actions (e.g. termination).

State attributes of a task, like "activated", "terminated", "terminatable" and so on, should be

global being obviously updated from the directly implied task (activation, termination), but also used by the creator (which is not necessarily the master) task (in order to verify the termination of the activation of the created tasks), or used by other tasks (rendezvous, termination, abort etc.).

All the information about queues and entries (e.g. for the representation of conditional entry calls) should be global too.

A compete definition of these structures can only be completed during the formal definition, and is out of the scope of this report

4.0 THE ATOMIC ACTIONS PROBLEMS

A formal definition of the concept of "atomic action" of a task can be given only in the framework of a fully mathematical model. However with reference to a model based on labelled transition system, we can think of an atomic action first as a labelled transition s.t. the intermediate states are not observable and hence not relevant to the overall semantics of a program.

On the contrary the beginning and the end of an atomic action mark the states in which a task can interact with the system, interfering and/or being interfered.
The problem in Ada is that the beginning and the end of an atomic action are not given simply by the various synchronization points or by the beginning and the end of a concurrent action. Due to the possibility of abortion, the case of shared variables and the obvious fact that the evaluation of expressions can involve the execution of subprograms and tasks, even many apparently sequential actions have to be split in more elementary actions in order to handle properly the concurrent interaction among tasks.

A second, now methodological requirement over the atomic actions is that their length should not be longer than the execution of a single Ada statement (or declaration), provided that we are interested in a syntax directed style in the description of the semantics of the language.

But in general we have that the effects of an Ada statement are too complex for being considered as atomic and should be specified again as a set of possible sequences of atomic actions.
Even looking at the manual we can observe that the effects of a statement (or declaration) are in general described by a sequence of smaller actions; this is obvious in the case of compound statements, but in general it happens so for "terminal" statements (and declarations) too (e.g. assignment, abort, exit, ...).

For example, in the case of an assignment, we have to evaluate an expression and a name in order to proceed with the update, and both the evaluations may involve (by means of function calls) an unlimited amount of activity. Then it seems reasonable, still from a methodological point of view, to follow the style of the manual in the description of the effects of a construct (decomposing statements and declarations in smaller pieces), at least <u>until</u> this description is driven by the <u>syntactic structure of the construct</u>.

On this ground, in the end atomic actions should not be longer than the elaboration of a "terminal" construct (e.g. names, literals, basic operators, ...).

Obviously, we still have to verify that it is correct to model such elaborations atomically, and for doing this we need to state some kind of requirements over the observable behaviour of these elaborations.

What we require from such elaborations, in order to be allowed to be considered atomic actions, is that they should not have observable "intermediate states" (in the sense that their intermediate states should not influence the behaviour of the rest of the system nor should be influenced by it). Non-atomic elaborations should be split again, until atomic actions are found.

We can observe that in some cases the mentioned elaborations are still too complex for being modelled as atomic. Indeed the concrete syntax of Ada sometimes hides long sequences of elaborations possibly having the same complexity of the whole program.
With our requirements, apart from some "hiding" constructs, most of the elaborations corresponding to "terminal" constructs, seems to be atomic. In fact the granularity of these actions already solves the problems of synchronization points or concurrent interactions.

The possibility of becoming abnormal, and hence completed prematurely, might influence the atomicity of an action; however though an abort statement can interrupt the execution of an action, it should not make observable the "intermediate states" of the action itself.
An interesting example of interference of the abort statement with an atomic action is illustrated by the update action. Indeed when a task becomes completed while updating a variable, it is specified from the manual that the value of the variable becomes "undefined". This explicit remark of the manual allows us to consider as atomic the update action (even in the case of updates of structured variables like arrays), because the effects of the interaction of this action with the rest of the system still does not depend on the set of the intermediate states reached by the action itself (even if it depends on the behaviour of the rest of the system) (the situation would have been completely different if some "partial update" might have occurred).

The situation is not so clear for many other actions ; for example it is not clear what might happen if an abort statement is itself prematurely abandoned because of another abort statement (might only a subset of the required tasks to become abnormal?).

Another issue is that some of these elaborations corresponding to "terminal constructs" might in general not be observable (for example the evaluation of a single name) and might be "packed" with other actions.

5.0 OTHER ISSUES

5.1 *Explicit Time*

A complete definition of Ada should describe all the time-dependent features of the language. These features are related to the existence of a predefined "CALENDAR" package, and to the existence of explicitly timed statements.
Some more sophisticated problems are related to the duration of other (not explicitly timed) statements.

The CALENDAR package provides a CLOCK function returning the actual value of the time (see LRM 9.6 (7)); obviously this is an implementation dependent feature and should be treated in a parametric way.
It seems reasonable that subsequent invocations of this function return increasing values of time; but this is not explicitly stated in the LRM.
Our approach can accomodate any of the official interpretation that can be taken in some future.

The effect of a single delay statement can be observable within a program, as it is illustrated by the following program fragment:

```
. . .
t := CLOCK;
delay (n);
newt := CLOCK;
. . .
```

the value of newt should be at least t+n .

We must note that in the CALENDAR package appropriate "+" and "<" functions are defined, allowing to sum a TIME value (returned from the CLOCK function) with a DURATION value (possibly used as parameter in a delay statement) and allowing to compare two different TIME values.

Analogously the effect of a delay alternative within a timed accept statement is observable, as it is illustrated by the following program fragment:

```
. . .
t := CLOCK;
select
  accept E
or delay (n); t' := CLOCK;
end select;
. . .
```

the value of t', if the delay alternative is executed, should be at least t + n .

Nevertheless we are aware that this is a particular interpretation of the manual, which really says nothing about the semantics of time; note however that the above interpretation is supported by the existence of some ACVC tests checking for the verification of the illustrated properties.

We believe that a formal treatement of the timed constructs should take into account these intuitive properties, for example modelling explicitly the current value of time.

5.2 *Parallelism*

We must observe that the LRM clearly states that an implementation is allowed to perform contemporaneously any group of eligible (nonexclusive) actions (9.0 (2)).

Moreover it is said that the duration (relative speed) of the actions is not specified (1.1.1 (12)).
Thus if it was not for the presence of constructs with an explicit reference to the priority feature, we can model a parallel execution by allowing, at each execution state of the system, any group of eligible actions to be performed in parallel. In a SMoLCS model this is dealt with by defining a free-parallel monitoring (see [Astesiano et al. 85 b] for an example).

Introducing priorities implies that at monitoring level we have some monitoring information related to task priorities and that parallelism is free except that for priority constraints.

However the only constraint that the language seems to state is related to the behaviour of a selective accept statement, when tasks with different priorities are queued (and hence eligible for execution if the corresponding entry is accepted) on different open entries.

5.3 *Implementation Dependent Aspects*

The implementation dependent aspects of Ada are of very different kinds.
Some of them are not explicitly implementation dependent features, in the sense that an implementation is not required to give an accurate description of them in some "appendix" of the manual, and are treated in the language as explicit forms of non-determinism (which an implementation is allowed to restrict, but which a program is not allowed to test). Notorious examples of this kind of "implementation dependent" aspects are the orders of elaboration of some constructs, the techniques for parameter passing, and so on. In these cases, even if an implementation is allowed to restrict the allowed non-determinism, a formal specification could not avoid to describe all the possible alternatives.

A similar example of the above mentioned nondetermism of the language is related to the concurrent aspects; each implementation can provide a particular scheduler, monitoring in its own way the relative speeds of tasks, and competitions in rendezvous. In this case a program is able to detect the implementation choices (at least in part), even if a formal specification could not avoid to describe all the possible alternatives in order to define the correctness or the uncorrectness of a program.

A completely different kind of implementation dependent aspects of the language are, on the converse, those aspects which should be explicitly described and fixed in some appendix. For example the definition of the type PRIORITY, DURATION, the values MEMORY_SIZE, MAXINT, the set of predefined numeric types, and so on. These aspects perhaps should be treated in a parametric way in the formal definition, because their non-determinism is not dynamic but fixed "a priori".

Another very different kind of implementation dependent features is related to the use of low-level facilities of Ada , e.g. association of entries with external interrupts, use of machine code insertion, mapping of objects at explicit ADDRESS values and so on. (It is not sure that this aspects should be modelled , and how). These aspects are not very interesting to be modelled in the formal definition.

A more precise report on these issues (including I/O problems) is still in preparation (see [Fantechi et al.] for more details).

6.0 CONCLUSION

The motivations of the choice of a model have been illustrated. In particular the advantages of a flat (single level) structure are explained.

Some hints on the treatement of timed constructs and other implementation dependent aspects of Ada have been given.

7.0 REFERENCES

[Astesiano 84] ASTESIANO,E. Combining an Operational With an Algebraic Approach to the Specification of Concurrency. To appear in *Proc. Workshop on Combining Methods* (Nyborg, Denmark, 1984), also in Cnet report n. 127 , December 1984.

[Blum 84] BLUM,K. An Abstract Systems Model of Ada Semantics.TRW Redondo Beach CA, August 1984.

[Astesiano et al. 85 a] ASTESIANO,E. MASCARI,G. REGGIO,G. AND WIRSING,M. On the parameterized algebraic specification of concurrent systems. *Proc. CAAP'85-TAPSOFT conference*, Berlin, Springer LNCS 185, March 1985.

[Astesiano et al. 85 b] ASTESIANO,E. AND REGGIO,G. A Syntax-directed approach to the semantics of concurrent languages .Preliminary report, May 1985.

[Plotkin 81] PLOTKIN ,G. A structural approach to operational semantics Lecture notes, Aarhus University, 1981.

[Berry 71]BERRY, D.M. Introduction to Oregano, *Proc. ACM SIGPLAN Symp. - Data Structures and Programming Languages*, Gainsville, Fla. Feb. 1971.

[Johnston 71] JOHNSTON, J.B. The Contour Model of Block Structured Processes, *Proc. ACM SIGPLAN Symp. - Data Structures and Programming Languages*, Gainsville, Fla. Feb. 1971.

[Bjorner et al. 80] BJORNER,D. AND OEST,O.N. Towards a formal definition of Ada LNCS 98, Springer 1980.

[Dewar et al. 83] DEWAR, R. FROELICH ,R.M . FISHER,G.A. AND KRUCHTEN,P. An executable semantic model for Ada, Ada/Ed interpreter Ada Project, Courant Institute, NYU 1983.

[Reggio 85] REGGIO,G. A proposal for an abstract storage model Working paper Ada_FD, April 1985.

[Fantechi et al.]FANTECHI,A. AND MAZZANTI,F. Notes on the implementation dependent

features Working paper Ada_FD, in preparation.

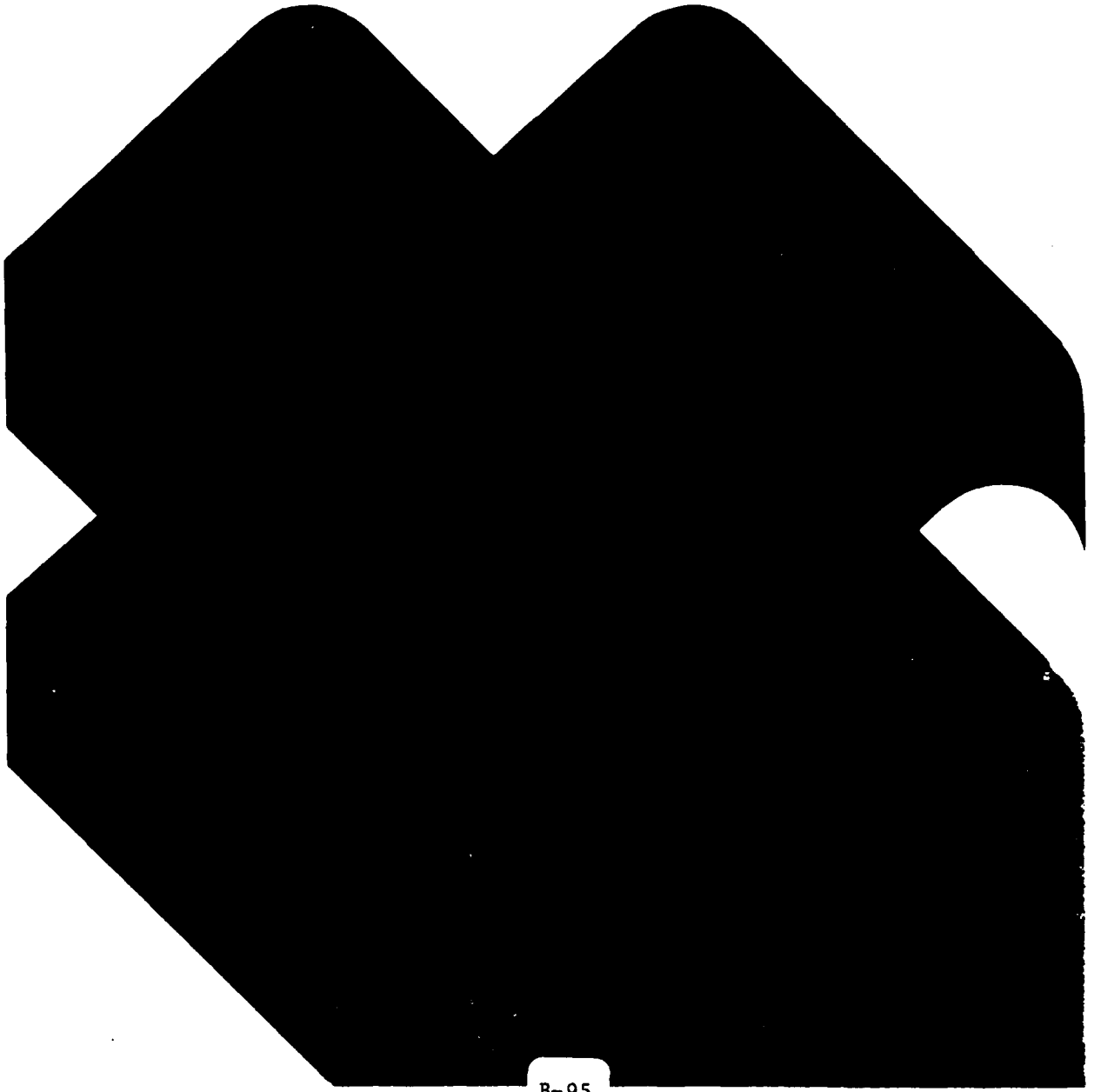
The Draft Formal Definition of Ada®

Commission of the European Communities: Multi-Annual Programme

FORMAL SPECIFICATION AND DEVELOPMENT OF AN ADA COMPILER-
A VDM CASE STUDY

Geert B. Clemmensen
Ole N. Oest

December 1983



*Ada is a registered trademark of the U.S Government (Ada Joint Program Office)

FORMAL SPECIFICATION AND DEVELOPMENT OF AN Ada* COMPILER

- A VDM CASE STUDY

Geert B. Clemmensen and Ole N. Oest

Dansk Datamatik Center
DK-2800 Lyngby
Denmark

ABSTRACT

The Vienna Development Method (VDM) has been employed by Dansk Datamatik Center (DDC) on a large-scale, industrial Ada compiler development project. VDM is a formal specification and development method in that it insists on the initial specifications and all design steps being expressed in a formal (mathematically based) notation.

This paper gives an overview of how VDM was used in the various steps of the DDC Ada project, and we guide the reader through the steps involved from the initial formal specification of Ada down to the actually coded multipass compiler. Finally we report on the quantitative and qualitative experiences we have gained, both as regards the technical suitability of VDM for the project and as regards the implications on software management and quality assurance.

1. Introduction

This section gives an overview of the Vienna Development Method (VDM) including its application in compiler development (subsection 1.1) and of the DDC Ada Compiler Project (subsection 1.2). Then section 2 describes how VDM was actually employed on the DDC Ada project taking into account the practical restrictions, some stemming from the kind of host and target computers, others stemming from the changing environment (three Ada Language Reference Manuals were issued during the project).

Section 3 examplifies the application of VDM in the development of the code generator, and finally section 4 reports on the experiences gained with VDM.

1.1 The Vienna Development Method

The Vienna Development Method was initially developed at the IBM Laboratory at Vienna in the early 1970's for the purpose of the definition of a large subset of PL/I [1], and the subsequent development of the corresponding compiler. VDM is based on the approach of denotational semantics, and should not be confused with the earlier work of the IBM Vienna Group, namely the specification language VDL, in which PL/I was specified in the late 1960's, and which relies on operational semantics. VDM uses a metalanguage known as "META-IV" [2] based on sugared lambda calculus [4] and Scott-Strachey domain theory [15]. But VDM is more than just a meta-language: a number of general approaches developed elsewhere has been incorporated into VDM, most notably stepwise refinement of functions as well as of data objects. VDM further contains a number of specialized approaches: in the area of programming language definition and compiler development VDM offers a specific set of guidelines thought of as a "cookbook" prescription for the work to be carried out [3]. As a part of reporting on our experiences, this paper explains which deviations from the "cookbook" we had to make, and why.

Generally software development proceeds as follows when using VDM:

A specification of the software to be developed is given in the form of a model, that is as operations (functions) on objects representing the input to and the internal state of the software, and yielding objects corresponding to the output and the changed internal state. The model is formal in the sense that it is expressed entirely in the meta-language, and it is abstract in the sense that it is free from details concerning the eventual implementation (functions are often defined implicitely rather than via an algorithm, and the actual representation of the objects is not considered at all. Objects can be abstract (e.g. recursively defined sets and mappings) with no counterpart in the implementation language). Classes of the objects involved (domains) are explicitly defined by so-called domain equations rather than implicitly defined (e.g. by axioms).

Then a series of more and more concrete specifications (called "designs") are worked out. Each design is derived from the preceding, more abstract specification in that either the objects, or the operations, or both are "refined" into corresponding objects and operations more close to the final implementation. All specifications are expressed in the meta-language, and the more concrete they become the more implementation details will be dealt with. Ideally the derivation of a more concrete specification is done formally

*) Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

by writing functions that "generates" the lower level objects, or by writing the so-called retrieve functions which, given the objects of a certain level, "retrieves" the corresponding objects of the higher level. It must then - in either case - based on these functions, be proved or argued that the derivation of the operations are correct. In practice, a less formal transformation from one level to the next takes place, cf. section 3, and as generally discussed in [12]. In certain application areas overall guidelines exist for the derivation of designs: however, most of the derivations one has to carry out are based on experience and skill. As regards the transformation of objects (e.g. mappings into tables), various standard examples exist. Refer [3].

VDM in compiler development proceeds as follows :

The departure point is a formal definition preferably in the denotational semantics style of the language to be compiled. The use of a formal definition of Ada as the basis for compiler construction is also advocated in [7]. Such a definition has three components in the case of Ada : A definition of the static semantics (SS), a definition of the dynamic semantics of the sequential constructs (DSS) and a description of the dynamic semantics of the parallel (tasking) constructs (DST).

The static semantics takes as "input" an Ada compilation unit represented in an abstract syntax AS1. The SS checks the corrrectness of the unit, and transforms it into another abstract syntax, AS2. In AS2 all information which is only relevant for the static semantics has been removed.

The formulae of the dynamic semantics assigns "meaning" to the compilation unit represented as AS2 objects.

AS1 and AS2 are based on an abstraction of the concrete syntax of the language being defined, as is also the DIANA intermediate language for Ada [13].

The front end compiler is derived from the static semantics, and the back end compiler (code generator) is derived from the dynamic semantics for the sequential constructs. The dynamic semantics for tasking constitutes the departure point for the tasking kernel in the run time system.

AS1 and AS2 will thus have their counterparts as intermediate languages in the compiler implementation. As the SS and DSS specifications will have to be split onto several passes, more intermediate languages will emerge during the design process. The specification of the code generator is called a compiling algorithm (CA), as it shows which code to generate for each construct in Ada (AS2).

Ideally a macro expansion step between the DSS and CA should be taken : The macro step generates "meta-language" rather than actual code, and allows for experiments with the actual run time system administration before details of the actual code is considered.

1.2 The DDC Ada Compiler Project

Dansk Datamatik Center (DDC) is involved in the development of an Ada compiler as a part of the Portable Ada Programming System (PAPS) project. The PAPS project is being carried out by Olivetti, Italy, Dansk Datamatik Center and Christian Rovsing A/S, Denmark, and will result in a programming environment initially hosted on and targeted for two 16 bit mini-computers, namely the Olivetti M40 and the Christian Rovsing CR80. The project includes a kernel operating system for Ada, various tools, an Ada compiler, and a high level machine for Ada. The project is partially funded by the European Community.

The host and target computers in question have imposed a number of restrictions on the project, the most severe being that the compiler should fit within 80 K bytes of code and 110 K bytes of data space. This has influenced the design of the compiler considerably: A multi-pass compilation technique has been chosen, with a total of 8 passes, and the tree-structured intermediate texts are linearized and scanned sequentially by each pass. The complete trees are thus not residing in internal memory; the symbol table, however, is placed in a software paged memory, administered by the compiler itself.

This design had some implications on the way VDM could be used on the project.

2. Employment of VDM on the DDC Ada Project

The goal of the DDC Ada project is the development of a portable Ada compiler written in (a subset of) Ada itself. Hence, a bootstrap tool is required. This tool, which is a source to source translator mapping Ada onto a medium level language SWELL, was also developed using VDM. This tool, called SLC-Ada (Source Language Conversion of Ada), was implemented in Pascal.

The two parts of the project, the development of SLC-Ada and of the Ada compiler, are treated separately, as they have quite different characteristics:

The development of the SLC-Ada translator could be considered as an internal effort of modest size. Little interaction with groups outside the DDC was necessary and the Ada subset chosen was stable during (and to a large extent after) the development phase so this sub-project could be managed with little effort and it could be carried out by a small group of 3 to 4 persons. This simplified the internal communication in the group. The sub-project was carried out in a little less than one year calendar time.

The development of the Ada compiler, however, was a large scale effort involving 10 to 14 persons over a three year period, and involving international cooperation on interfaces within the compiler as well as between the compiler and the environment. Further the Ada language fluctuated quite heavily during the project period (probably more from a compiler writers point of view than as seen from the average user of Ada). Not only did we see three issues of the Ada reference manual during the project, but we also saw inbetween these issues various - mutually and internally inconsistent - interpretations of the Ada reference manual (the Softech Implementor's guide, the Ada Question/Answer mechanism on the ARPA-net). This influenced the project to a large extent, and implied a rather pragmatic use of VDM, as the goal of the project was to come out with an up-to-date compiler, rather than to maintain a coherent set of formulae through all development steps. On the other hand, the obligations to the "outside world" required that a high and consistent level of documentation had to be maintained. Thus a careful balance had to be made.

It should be mentioned here that we had no tools available to support the development of the formal specifications, to check their consistence or to help in the refinement steps. VDM was (and still is) a paper and pencil method although steps are being taken now to develop support tools.

SLC-Ada

The subset was chosen according to experience with earlier program and compiler development. The guiding factors were:

1) the Ada subset was to be used as implementation language for a 100.000 lines project,

2) straightforward implementability of the selected features.

The static semantics of the subset was described in META-IV, and the dynamic semantics was described by giving a compiling algorithm mapping the subset into SWELL. As a parallel effort outside the PAPS project a formal description (static and dynamic semantics) of SWELL was worked out. The formal description of the subset was intended to form the formal specification of the compiler and was hence written with some thoughts about implementation issues.

The translator was coded by a rather direct, manual transformation or rewriting of the static semantics and the compiling algorithm in Pascal. This resulted (in addition to the scanner/parser) in one pass handling the static semantics, and one pass performing the source translation. Due to memory restrictions it later became necessary to split the static semantics pass into three separate passes. This was done without introducing different intermediate languages. All passes (except the scanner/parser) work on the same intermediate language (and repeats certain operations). The mechanisms for the separate compilation, the scanner/parser, the run time system and the Ada linker were developed by traditional means.

2.2 The Ada Compiler

The development proceeded in this case through four steps:

1: Development of a formal specification of Ada. The static semantics and the sequential part of the dynamic semantics are specified in the denotational semantics style, whereas tasking is specified by an operational model [8], [9], [10], [11].

2: Development of a formal specification of the compiler parts.

3: Development of a more detailed formal specification of certain compiler components and passes.

4: The Ada program structure is decided upon; the specifications are broken into Ada packages, and implemented in Ada.

Application of steps 2 and 3 to the front end compiler involved:

step 2:

- identifying items governing the grouping of static checks (dependence on completeness of symbol table contents, on degree of overloading resolution, on evaluation of static expressions etc.),
- classification of the static checks (based on the formal specification of Ada and on the items identified above),
- distributing the static checks to the passes based on a topological sorting of the checks,
- formal specification of the passes and intermediate languages,
- defining the intermediate languages between the passes of the front end,
- specifying for each pass the transformation from the input intermediate language to the output intermediate language,
- specifying a symbol table handler.

step 3:

- for some parts derivation of a more implementation oriented specification; for the remaining parts the formulae of step 2 apply.

The static semantics part of the formal description of Ada has some resemblance with a one pass compiler. No formal methods exist to derive a specification of a multi-pass compiler front end from such a definition. However, by employing systemacy as described under step 2, we obtained a compiler front end specification which turned out to be of a very high standard: It contained rather few errors, and they were all easy to correct. However, as step 1 (and later 2) became obsolete with the new issues of the Ada language reference manual, the specification of step 3 was updated, proof-read and compared directly with the text of the new manual. For each formula in the specification, the corresponding test of the manual was marked with the number of the compiler pass, which handled that text. Finally it was checked that all of the manual had been marked up, and necessary changes to the formulae were carried out.

Application of steps 2 and 3 to the back end compiler involved:

step 2:

- formal specification of an "overall" compiling algorithm mapping the output of the front end directly into the A-code instruction set of the high level target machine [6]. Note, that the intermediate language between the front end and the back end stems from AS2; hence it had only to be refined in steps 2 and 3, not to be defined.

step 3:

- based upon this: decision on intermediate language levels and structure,

- formal specification of the resulting three back end passes and two intermediate languages.

The back end development is examplified in section 3. The specifications of step 2 above is based on step 1 and the informally described changes of Ada, which took place during the development of step 2. Step 3 was developed in a similar way. Only the step 3 definitions are maintained up to date with respect to the current Ada definition.

The scanner/parser and the separate compilation handler were developed using traditional methods.

In order to obtain intermediate milestones, four implementation levels were defined where each level implements more and more of Ada. This division into levels was done based on the formal specification, and largely only after the development of all of the specifications. Each level has been tested thoroughly, both by the original developers and by an independent group.

3. A VDM Exercise

As described elsewhere in this paper, VDM is based on an initial formal specification onto which a sequence of refinement steps are applied in order to reach the final implementation. In this section, the reader is guided more or less informally through an exercise in VDM by showing how the dynamic semantics of a specific Ada construct is specified, refined and implemented (in a subset of Ada).

To introduce some of the terms used in the following, the compiler structure is shown :

Ada -> Front End -> IML_6 -> Back End -> A-code

Within the back end the following structure exists:

IML_6->Pass 6->IML_7->Pass 7->AA-code->Pass 8->A-code

IML_6 is a tree structured intermediate language which is comparable to DIANA [13] in level, but compacted and simplified. IML_7 is also a tree structured intermediate language, but is aimed at code generation for any class of target machine. A-code is the code for a virtual stack machine called the A-machine [6]. Abstract A-code (AA-code) is a suitable abstraction of A-code which eases the code generation, that takes place in pass 7, and makes it possible to choose among different implementations of the A-machine.

The Ada construct used as an example is the *object declaration* [14 section 3.2] :

object declaration ::=
identifier_list : [*constant*] *subtype indication* [:= *expression*]

subtype_indication ::= type_mark [*constraint*]

As mentioned earlier, the formal specification of Ada has two main components, namely the Static and Dynamic Semantics. An abstract syntax of Ada, called AS1, forms the input to the Static Semantics which also contains a transformer producing the abstract syntax of the Dynamic Semantics, called AS2. In AS2 the construct is modelled as follows :

Object-del :: Var-id-set [*CNST*] *Subtype-def* [*Expr*]
Var-id :: TOKEN
Subtype-def :: Type-mark [*Constr*]

Notice how close the domain specification is to the original Ada syntax given above.

The elaboration of an *Object-dcl* can now be formally specified as :

```
elab-Object-dcl(decl)env =

let mk-Object-dcl(vids,cnst,st-def,iexpr)=decl in
  (def st : elab-Subtype-def(st-def)env;
    (cnst = CNST ->
          elab-Const-dcl(vids,st,iexpr)env,
     T          ->
          elab-Var-dcl(vids,st,iexpr)env))
```

type: *Object-dcl -> (ENV => ENV)*
pre: The elaboration of the subtype definition has no side effects

```
elab-Var-dcl(vids,st,iexpr)env =

(def varenv : get-vardens(vids,st)env;
 (iexpr = nil ->
    for all vid ∈ vids do
      (def ival : get-init-VAL(st)(CREATE)env;
       assign(s-VarLoc(varenv(vid)),ival));
    return(varenv)),

  T          ->
    for all vid ∈ vids do
      (def ival : eval-Expr(iexpr)(env+varenv);
       subtype-check(ival,st)env;
       assign(s-VarLoc(varenv(vid)),ival));
    return(varenv)))
```

type: *Var-id-set Subtype-den [Expr] -> (ENV => ENV)*

The elaboration of an *Object-dcl* consists of elaboration of the subtype definition (yielding a so-called subtype denotation) and a new local environment (*elab-Const-dcl* or *elab-Var-dcl*) in which the objects are introduced.

The elaboration of a *Var-dcl* consists of creation of a local environment in which the objects are introduced and the evaluation and assignment of either implicit or explicit initialization expressions.

One important issue of the specification given above is that all kinds of objects (arrays, tasks, simple) are treated uniformly which compacts the specification and eases the reading considerably.

When the formal specification from the Dynamic Semantics is to be refined (including both domain and operation refinements) into a so-called compiling algorithm (code generator) specification, a number of important issues must be addressed in order to guide the refinement process.

Examples are :

- how to implement the various kinds of objects
- elimination of checks where possible
- optimization of repeated expression evaluation

In our implementation we have decided to distinguish between the following object kinds :

- array
- record
- task
- access
- remaining and simple objects

In the following we will concentrate on the simple objects.

In order to get some hints on how to direct the refinement and implementation process it was decided to work out an experimental refinement step, transforming AS2 directly to pseudo A-code. This so-called compiling algorithm sketch resulted among other things in the notion of predicates. Predicates are truth values attached to the various nodes of IML_η and they express certain properties about the sons of the nodes (i.e. they guide the code generation).

The actual refinement steps can now be given :

AS2 -> IML_6 : This refinement step is a step in the design process and is not implemented, it merely consists of a concretization of AS2 into IML_6. The step is called a domain refinement.

IML_6 -> IML_η : In this refinement step the various objects (also types, subprograms, operators etc.) are classified into the appropriate kinds and predicates are evaluated. Essentially this refinement step is also a domain refinement, although not normally covered by the term domain refinement. The step is implemented as pass 6.

IML_η -> Abstract A-code : In this refinement step the high level tree structured intermediate language IML_η is transformed to a linearized sequence of Abstract A-code instructions. This step is the operation refinement step, and is implemented as pass 7.

Abstract A-code -> A-code : This last refinement step takes the Abstract A-code and produces the final A-code. The step can be viewed as a post domain refinement step which concretizes the domains of the Abstract A-code.

Because of the rather voluminous specifications of all the refinement steps, the previous example will only be shown specified in the refinement step IML_η -> Abstract A-code.

In IML_η the previous example is modelled as :

```
SimpleObjectDecl :: OBJECT-KIND ObjIdl
                                [SimpleConstr] [Expr]
OBJECT-KIND      = ShortIntg | Intg |
                   LongIntg  | .....
ObjIdl           :: len:INTG DESCR-ADDR+
```

Predicates :

SimpleConstr-StaticBounds *Expr-NoSideEffects*
SimpleConstr-NoLowerCheck *Expr-NoSubtypeCheck*
SimpleConstr-NoUpperCheck *Expr-NoLowerCheck*
SimpleConstr-NotNullRange *Expr-NoUpperCheck*

The *Var-id-set* of AS2 has now been converted to a list of object identifiers which essentially is a list of symbol table references. Predicates, expressing properties about the constraint and the initialization expression, are also evaluated and made available.

The *elab-Var-dcl* elaboration procedure has now become a so-called Compiling Algorithm formula and is named *C-SimpleObjectDecl* (shown below). Constants are treated no different than variables in this implementation, but other implementations may choose different refinement directions and hence a different compiling algorithm and implementation.
It should be noted how close the compiling algorithm formula is to *elab-Object-dcl* and *elab-Var-dcl*, but it is also clear that the compiling algorithm formula is more or less straightforward to implement compared to the elaboration formulas of the dynamic semantics. The actual implementation of the compiling algorithm formula, in a subset of Ada, is shown on the page following the formula.

Summarizing the steps involved :

1) A suitable high level abstract syntax of Ada, AS2, is defined and the dynamic semantics is formally specified
2) AS2 is refined into IML7 using so-called domain refinements
3) The Dynamic Semantics is refined into a Compiling Algorithm using so-called operation refinements
4) The Compiling Algorithm is implemented

Rudimentary annotations to the formula :

05-10 : If a constraint is given it is compiled and storage is claimed and associated to the symbol table handle *ta*. The evaluated predicates are fetched and used for generating the optimal code for the constraint.

12-16 : If no initialization expression is given, storage is allocated and the associated addresses are stored in the symbol table (via the *DESCR-ADDRs*). If storage allocation is to be done by pushes, the stack pointer is incremented resulting in undefined initialization values.

18-36 : The storage address of the constraint descriptor is extracted from the symbol table and the initialization expression is evaluated the required number of times and checked against the constraint before assigned.

```
01  C-SimpleObjectDecl(am,decl) =
02    let mk-SimpleObjectDecl(objkind,objidl,oconstr,oexpr) = decl in
03    let mk-ObjIdl(len,dal) = objidl in
04    let ta = e-TYPE-ADDR(dal[1]) in
05      (oconstr ≠ nil ==>
06           C-SimpleConstr(am,oconstr,ta,objkind,
07                                    SimpleConstr-StaticBounds(decl),
08                                    SimpleConstr-NotNullRange(decl),
09                                    SimpleConstr-NoLowerCheck(decl),
10                                    SimpleConstr-NoUpperCheck(decl)))
11        ^
12      (oexpr = nil ->
13           { let st-addr = get-obj-addr(objkind) in
14               insert-obj-addr(dal[i],st-addr) | 1 ≤ i ≤ len }
15             ^
16           (am = IMMEDIATE ==> Alloc_on_stack(len * size(objkind))),
17       T
18           let ea = extract-constr-addr(ta) in
19             (Expr-NoSideEffect(decl) ->
20                  C-Expr(oexpr,objkind)
21                    ^
22                  (-Expr-NoSubtypeCheck(decl) ==>
23                          Check_range(objkind,ea))
24                    ^
25                  C-AssignInitExpr(am,len,dal,objkind),
26              T
27                  { C-Expr(oexpr,objkind)
28                    ^
29                    (-Expr-NoSubtypeCheck(decl) ==>
30                            Check_range(objkind,ea))
31                    ^
32                    let st-addr = get-obj-addr(objkind) in
33                      insert-obj-addr(dal[i],st-addr)
34                        ^
35                      (am = DEFERRED ==>
36                              Pop(objkind,st-addr)) | 1 ≤ i ≤ len }
37
38  type: AllocMode  SimpleObjectDecl  ==>  AA-code
```

```
procedure C_SimpleObjectDecl is

   SimpleConstr_StaticBounds : boolean := iml7_get_pred(1);
   SimpleConstr_NotNullRange : boolean := iml7_get_pred(2);
   SimpleConstr_NoLowerCheck : boolean := iml7_get_pred(4);
   SimpleConstr_NoUpperCheck : boolean := iml7_get_pred(8);
   Expr_NoSideEffects        : boolean := iml7_get_pred(16);
   Expr_NoSubtypeCheck       : boolean := iml7_get_pred(32);

   objkind  : OBJECT_KIND          := iml7_objkind;
   p_odd    : obj_decl_descr_ref := uc_obj_decl_descr(sth_access(iml7_op.(1), r));
   od       : obj_descr            := uc_obj_descr(sth_access(c_odd.all.idescr, r)).all;
   ca       : ConAddress;
   iml7_pos : iml7_position;

begin
   iml7_in;
   if iml7_next /= nil then
      C_SimpleConstr(em, cc.obj_type, objkind, SimpleConstr_StaticBounds,
                                               SimpleConstr_NotNullRange,
                                               SimpleConstr_NoLowerCheck,
                                               SimpleConstr_NoUpperCheck);
   end if;

   iml7_get_pos(iml7_pos);
   iml7_in;        -- descr

   if iml7_next = nil then
      get_and_insert_obj_addrs(objkind);

      if AC_ap = IMMEDIATE then
         emit_c1(aa_alloc_on_stack, iml7_dal_len * size_of(objkind));
      end if;
   else
      ca := uc_simple_type(sth_access(od.obj_type, r)).all.constr_addr;

      if Expr_NoSideEffects then
         C_Expr(objkind);

         if not Expr_NoSubtypeCheck then
            emit_c5(aa_Check_range, objkind, ca);
         end if;

         C_AssignInitExpr(objkind);
      else
         for i in 1..iml7_dal_len
            loop
               if i > 1 then
                  iml7_set_pos(iml7_pos);
                  iml7_in;
               end if;

               C_Expr(objkind);
               if not Expr_NoSubtypeCheck then
                  emit_c5(aa_Check_range, objkind, ca);
               end if;

               p_odd := uc_obj_decl_descr(sth_access(iml7_dal(i), a));
               get_obj_addr(objkind, p_odd.all.obj_addr);

               if AC_ap = DEFERRED then
                  emit_c5(aa_Pop, objkind, p_odd.all.obj_addr);
               end if;
            end loop;
      end if;
   end if;
end C_SimpleObjectDecl;
```

4. Experience Gained

Section 4.1 presents some quantitative observations as regards the SLC-Ada sub-project and the main Ada compiler project. Section 4.2 contains some technical experience mostly concerning the deviations from "strict" VDM, and section 4.3 contains our experience as seen from a software management point of view.

4.1 Quantitative Observations

SLC-Ada Sub-Project

Component	Formula lines	Source lines	Hours
Scanner and Parser		3200	472
Static Semantics and Front End	1400	11500	1365
Compiling Algorithm and Code Generator	1300	6700	1285
Misc. (Library System, Linker, Run Time System, Utilities)		8500	368
Functional Test by QA Staff		3400	178
User Documentation			67
Total	2700	33300	3735

These figures include the design of the Ada subset and the development of the folowing informal documents: Design Specification (44 pages), Informal Program Specification (39 pages) and User's Guide (68 pages).

The initial estimate was 1860 person hours. The major reason for the overrun was that the complexity of the Ada subset was underestimated. Initially it was considered to be of the complexity of Pascal.

Ada Compiler Project

Component	Formula lines	Source lines	Hours
Static Semantics, Scanner, Parser and Front End	24000	55000	12400
Dynamic Semantics, Compiling Algorithms and Code Generator	20000	62000	9700
Separate Compilation Handler, Multi Pass Administrator, Supporting Packages		40000	2700
Misc.(SLC-Ada, other tools)	2700	66000	4200
User Documentation			400
Functional Test by QA staff ++)		7000	950
Other QA Work			1850
Overhead +)			11500
	46700	230000	43700

+) This includes: Management, meetings with partners and other implementors, conferences, work in Ada Europe on language review and standardization, computer operation.

++) As the official Ada Compiler Validation Capability test suite was used we had only to develop a few test programs.

The figures above include the development of the following documents, totalling 1200 pages not including the formulae: Requirements specification, functional specification, global design, detailed design including intermediate languages and symbol table, externally available interfaces, test reports. Further is included feasibility studies of intermediate languages used elsewhere, progress meetings, review meetings. All figures are approximate, as the project is not completed at the time of writing.

The initial estimate was 32,000 person hours, and that around 100,000 lines of code had to be developed (excluding the SLC-Ada system). The reasons for the overrun hasn't been completely analyzed, but among the reasons are: The Ada language changed during the project, the complexity was higher than estimated, the development took place on new hardware and on a pre-release of a new operating system.

4.2 Technical Experience

This section presents the areas where we had to deviate from "strict" VDM. However, it should be noted that VDM users are pragmatic rather than dogmatic, so that it is considered perfectly acceptable to adapt VDM to specific needs!

- Transformation of one step into the next was done systematically, but informally. No proofs of correctness were given. It is not feasible

to carry out proofs of correctness without tools which can aid the proofs: even with such tools the task might turn out to be very large.

- The formulae developed in the first two steps were not maintained up to date with the changing Ada language. Hence the complete line of documentation from the formal description of Ada down to the implemented compiler is lost. There are two major reasons for this: (1) the lack of tools makes it extremely difficult to maintain formal specifications of the size of the Ada project, and (2) maintenance of the formal specification of Ada is a major task in itself, taking the many changes and (still unresolved) semantic problems in Ada into account. In a compiler one can take certain decisions as regards the implementation of the semantics of Ada - this cannot be done in a formal specification.

- Development of a derivation step was based on the formal specification of the previous step and the informal description of the changes which had occurred to Ada in the meantime.

- A macro-expansion step between the compiling algorithm and the specification of the code generator was omitted. The macro expansion would have allowed for experiments with the storage layout at run time and with the run time administration. However, omission of the step was (partly) justified with the fact, that DDC was not directly involved in the development of the A-machine.

4.3 Software Management Experience

- Management of the project benefits, because each project member knows how the work should be done.

- The project status is more transparent due to the various intermediate milestones which have to be formally specified. Progress can be measured.

- The implementation can be divided into levels, or intermediate milestones, in a secure way based on the formal specifications. There is no risk that the resulting lower level subset compilers cannot be extended to full Ada, as has been seen on other projects.

- Based on experience from SLC-Ada and the formal specifications of the compiler passes, reasonably good estimates of the final program size and resource requirements can be made. However, it became evident that the experience from the earlier DDC CHILL compiler project could not be applied. This indicates that the actual style and level of the formal specifications are rather personal, in that they depend on the authors. Hence the amount of work in deriving implementations depends on the individuals involved.

- Adding staff with VDM experience to the project poses no problems. Moving staff from one part of the project to another poses no problems. Such staff changes are feasible in the specification phase as well as in the implementation phase.

- Adding staff in the implementation phase with little or no VDM experience (but with an introductory course to VDM) is not feasible. In such cases the staff should participate also in the specification phase, mainly for the purpose of education and motivation.

- Strict (rigorous) use of VDM is not feasible on a project of this size and nature: partly due to the size of the specifications and programs, partly due to the changing requirements (here the changes of Ada). Management must be able to deviate from strict VDM by giving in on formal derivations, on proofs/arguments of the correctness of the derivation steps, and by omitting certain derivation steps (e.g. macro-expansion specification between the compiling algorithm and the actual code generator). The advantage of VDM thus becomes that of enabling formal and precise definitions of each step and the associated interfaces. More rigorous derivations require software tools (transformation processors, proof and verification tools).

- The development of a formal definition of Ada as the first step gave a very valuable insight into Ada, and it made it easy for the persons involved to ascertain the consequences of the various changes of Ada for the compiler. However, it is not possible to derive in any formal way the specification of a multi-pass compiler from the Ada specification.

- Due to the complete formal specifications, reasonably final interface definitions (e.g. intermediate languages) can be given at a rather early stage. Hence, new staff members can be added for parallel work without much introduction.

- Focusing entirely on the Ada language semantics in the early phase hampered communication with other implementors who were more concerned with implementation details of various specific constructs. These implementors had still to discover and understand the more fundamental issues and problems.

- Development including management of formal specifications of a size comparable to that of the formal definition of Ada is hardly possible without the support of software tools (cross checking formulae, cross-referencing).

- Quite a large number of trivial errors in the specifications were not found until they were

detected in the corresponding code. Such errors could be detected in an earlier stage by proper VDM-tools.

- Maintenance of large formal specifications is not feasible without tools, unless the original developers are available for the maintenance.

- The lack of VDM tools makes production and maintenance of the documentation very expensive. The SLC-Ada documentation has been maintained only by marking the changes in pencil in the original documents. However, this approach is not satisfactory if the documents have to be used by persons other than the authors.

- In the SLC-Ada case the static semantics specification proved a useful reference document, which was frequently used to settle quickly any debate about the contents and meaning of the subset.

- 12 mostly minor errors were found in the SLC-Ada during the functional test carried out by the DDC Quality Assurance manager. None of the errors required changes in the initial design or implementation strategy. The program under test consisted of 30.000 lines. 18.000 of these were developed by use of VDM.

- The number of errors found after delivery of the SLC-Ada was very low. Less than 0.5 percent of the initial development time was used on maintenance and extensions of the subset.

- The Quality Assurance function could be applied at an early point in time: The formal specifications was scrutinized on a sample basis by the Quality Assurance staff, who mainly focused on critical areas as the symbol table building and application, and on the interfaces.

- The Quality Assurance staff must incorporate VDM skills (at least) of the level of the development team.

- The time schedule laid down in the original work plan of 1980 has been followed by and large; according to this schedule the compiler and run time system should be operational in September 1983. The date achieved was May 1983 for the level 1 subset, August 1983 for level 2, and February 1984 for full Ada.

- The ressources estimated in the original work plan in 1980 were insufficient; the overrun amounted to 37 per cent. Hence we could only keep the time schedule by adding staff to the project. VDM here helped to make this fairly easy as discussed above.

5. Conclusion

Our overall conclusion is that the project could not have been carried out to the achieved level of quality within the time frame available without the use of VDM. Comparisons with other methods cannot presently be made due to lack of data from similar large-scale projects carried out with similar formal methods. However, most, if not all, other formally based methods are too rigorous to allow for practical use - hence (part of) the advantages we have gained from VDM cannot necessarily be projected onto other methods as these will not be able to handle projects of the size and complexity of the DDC Ada project. A discussion of various methods based on experience from smaller projects is available in [5], whereas VDM has been given a critical review in [12].

6. References

[1] H. Bekić, D. Bjørner, W. Henhapl, C.B. Jones and P. Lucas: *A Formal Definition of a PL/I Subset*, IBM Vienna, TR25.139, Dec. 1974.

[2] D. Bjørner, O.N. Oest (eds.): *Towards a Formal Description of Ada*, Lecture Notes in Computer science, Vol. 98, Springer Verlag, 1980.

[3] D. Bjørner, C.B. Jones: *Formal Specification and Software Development*, Prentice-Hall International Series in Computer Science, 1982.

[4] A. Church: *The Calculi of Lambda-Conversion*, Annals of Math. Studies, 6, Princeton University Press, N.J., 1941.

[5] B. Cohen, M.I. Jackson: *A Critical Appraisal of Formal Software Development Theories, Methods and Tools*, ESPRIT preparatory study, STL, June 1983.

[6] L. Ibsen, L.O.K. Nielsen, N.M. Jørgensen: *A-Machine Specification*, ADA/RFM/0001, Christian Rovsing A/S, March 1983.

[7] V. Donzeau-Gouge, G. Kahn, B. Lang, B. Krieg-Brueckner: *On the Formal Definition of Ada*, Rivista Di Informatica, Vol. X, N.1, January-March 1980.

[8] G.B. Clemmensen, H.H. Løvengreen: *Portable Ada Programming System, Dynamic Semantics, Description of Ada Tasking*, DDC, Nov. 1981.

[9] J. Jørgensen: *Portable Ada Programming System, Ada Static Semantics, AS1 → AS2 Transformation*, DDC, Feb. 1982.

[10] H. Bruun, J. Bundgaard, J. Jørgensen: *Portable Ada Programming Syster, Ada Static Semantics, Well-formedness Criteria,* DDC, March 1982.

[11] J.S. Pedersen, P. Folkjaer, I.Ø. Hansen: *Portable Ada Programming System, Dynamic Semantics, Description of Sequential Ada,* DDC, March 1982.

[12] S. Prehn, I.Ø. Hansen, S.U. Palm, P. Gøbel: *Formal Methods Appraisal, Part II, A Critical Examination of VDM,* DDC, June 1983.

[13] *DIANA Reference Manual, Revision 3,* TARTAN Laboratories INC, Febr. 1983.

[14] *Reference Manual for the Ada Programming Language,* ANSI/MIL-STD 1815A, January 1983.

[15] J.E. Stoy: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* MIT Press, 1977.

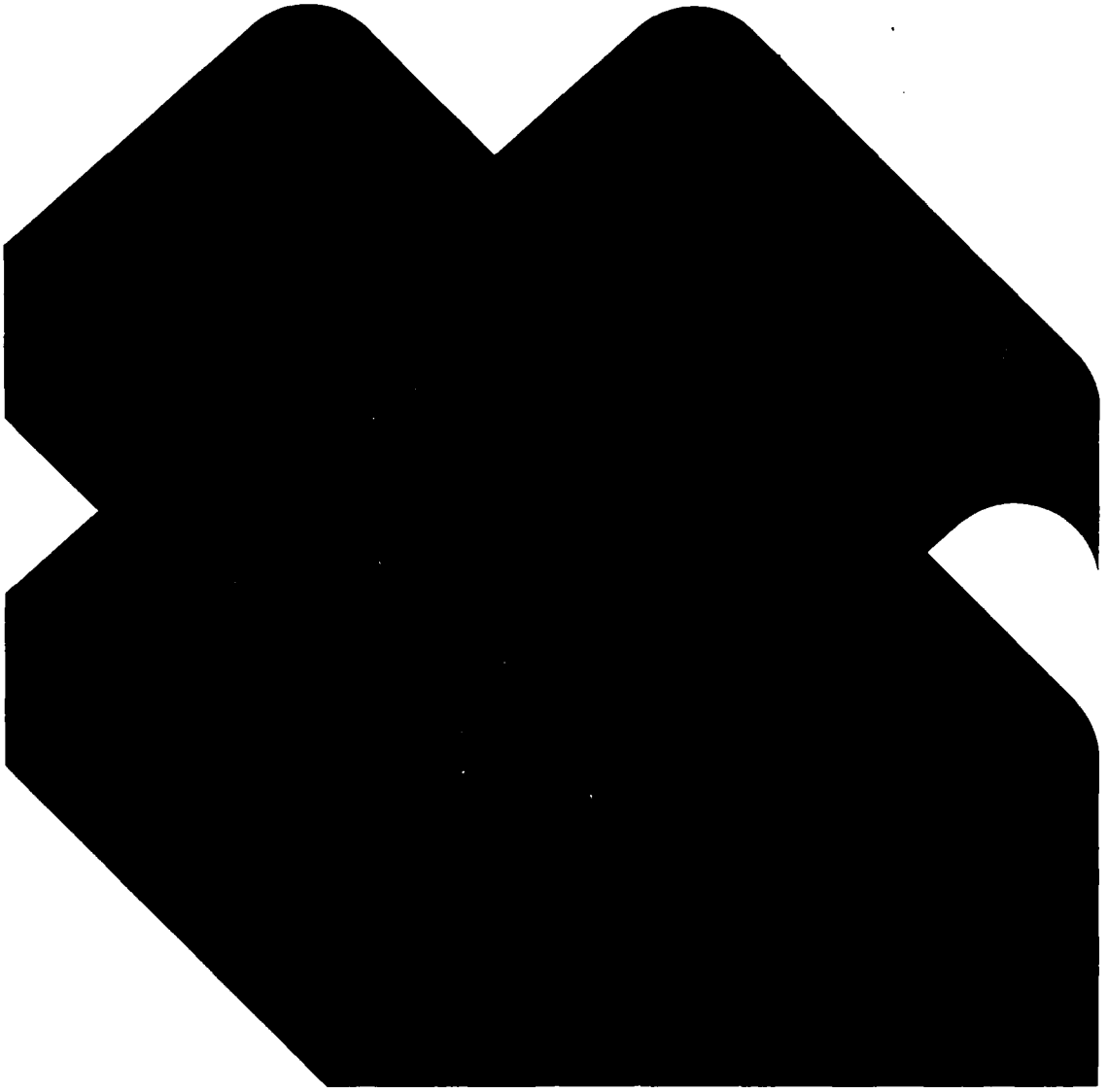
The Draft Formal Definition of Ada®

Commission of the European Communities: Multi-Annual Programme

The Rôle and Scope of the Formal Definition of Ada

Dines Bjørner

September 9, 1985



*Ada is a registered trademark of the U S Government (Ada Joint Program Office)

DOCUMENT HISTORY

(1) Version 0 of this document was very preliminary. It has not been internally reviewed among the Ada FD project participants.

(2) Version 0 was being externally distributed on a courtesy basis. It was not to be further distributed outside the project partners. Receivers were kindly asked to submit comments before 15 August 1985.

(3) Version 0 was subject to a write-in internal review. This internal review started 23 July and ended 15 August 1985.

(4) Version 1 resulted from this write-in review. It is now subject to a pre-external review: 23 August - 1 Sept 1985.

(5) There will be no external review of this deliverable.

(6) Version 2 should result from this formal internal review. It will then be submitted to the CEC, 9 Sept 1985.

(7) The CEC will review this version 2 on 1 Oct 1985.

(8) Further versions are expected to be produced throughout the project life.

PROJECT SPONSOR

This report represents work which is fully funded by the CEC (Commission of the European Communities) under the Multi-Annual Programme in the Field of Data Processing, Project No. 782: "The Draft Formal Definition of ANSI/MIL-STD 1815A Ada".

ABSTRACT

This document defines:

(a) Relevant Ada programming language issues,
(b) what is meant by a formal definition (FD),
(c) the various user groups of an Ada FD, and
(d) the uses these groups may have of such an FD.

From the extensional requirements (sects.2-3) that these users expect an Ada FD to fulfil, and from the state-of-the-art of formal definition techniques and methods (sect.4), we then derive the basic intended characteristics of the particular Ada FD to be constructed in this project, first ideally (sect.5), then realistically (sect.6).

This document is to serve as part of the final, full documentation constituting the Ada FD.

The rôle of this document is twofold:

(I) To serve as a "yardstick" with which to "measure" the conformance of the intermediate and final results of on-going Ada FD project work w.r.t. the perceived rôle of the Ada FD, and

(II) as one of several kinds of introductions to the Ada FD project.

The present, initial, version will differ slightly from a final version in that it addresses mainly document rôle (I), whereas the final version should address rôle (II).

HOW TO READ THIS DOCUMENT

Sections 5 and 6 contain the core of this report.

CONTENTS

0. PROJECT OVERVIEW

We briefly give a set of contextual facts concerning "The CEC MAP Project: The Draft Formal Definition of ANSI/MIL-STD 1815A Ada" henceforth referred to as the Ada FD project. For understanding the unpleasantly heavy use of mnemonics, please refer to Appendix A.

0.1 Background

The Ada programming language is described informally in the so-called Language Reference Manual, LRM, also known as the ANSI/MIL-STD 1815A standard.

Many Ada compilers (several academic and several industrial) have been, or are being developed, in USA and Europe (West and East) - world-wide. Many, including some commercial compilers, are labelled Ada, but compile subsets of, or extensions to Ada. Ada appears to be destined for extensive use in educational, commercial, industrial, and military contexts.

There is an obvious need for an Ada standard with no deviations: subsets, extensions, errors, or mis-interpretations.

The US DoD was, from the very beginning, clearly aware of this. And the CEC quickly established industrial projects not only aimed at producing European Ada compiler products and competence, but also, on a broader scale, at acquiring deep and widespread insight into all aspects of Ada. Thus, the CEC, in 1980, established a number of very active so-called "Ada Europe" working groups.

The present project must be seen as an outgrowth from several years of often deeply technical and theoretical discussions, especially in the Ada-Europe Working Group on Formal Semantics, and the working group on Formal Methods.

0.2 Purpose

The purpose of the Ada FD project is to produce, during 1985-1986, a draft formal definition of the language as defined by the language Reference Manual ANSI/MIL-STD 1815A Ada.

We list the major deliverables:

I: A Formal Definition, referred to as the Ada FD, of ANSI/MIL-STD 1815A Ada.

II: A precise definition, referred to as the Ada FD MTL, of the definition Methods, Techniques, and Languages (notations) used in producing the Ada FD.

III: A detailed, comprehensive cross-reference, referred to as Ada FD/LRM, between the Ada FD and the ANSI/MIL-STD 1815A LRM.

IV: An Ada FD Primer introducing the Ada FD, in careful overviews and details, independent of the LRM.

V: Computerized, reasonably portable tools for reading and manipulating (i) the Ada FD, (ii) the Ada FD MTL, (iii) the LRM and the Ada FD/LRM, and (iv) the Ada FD Primer, i.e. all essential documents produced by this project.

0.3 Project Partners

The Ada FD project is carried out under an almost fully paid contract to the CEC jointly by Dansk Datamatik Center (DDC) (Denmark) and CRAI (Consorzio per la Ricerca e le Applicazioni de Informatica) (Italy). DDC is the main contractor.

In this Project DDC makes use of consultants (Prof. Hans Bruun and Hans Henrik Løvengreen) from the Department of Computer Science at the Technical University of Denmark.

CRAI has sub-contracted certain parts of these project parts to the CNR-IEI in Pisa (Istituto di Elaborazione della Informazione of the Italian Consiglio Nazionale delle Ricerche), and otherwise makes use of consultants from the Universities of Pisa and Genoa (Prof. Ugo Montanari, Inst. of Informatics, Pisa, and Prof. Egidio Astesiano, Math. Inst., Genoa).

DDC has more than 5 years of experience in formal definitions (mainly the CHILL and Ada programming languages), in extensive Ada programming (more than 1/3 million lines of Ada), and in systematical development, from formal definitions, of production quality compilers for CHILL and Ada.

CRAI, with its sub-contractor and consultants, has played a major rôle in the Italian Consiglio Nationale delle Ricerche project Cnet: a formal programming methodology and software engineering project for distributed programming and computing (Campus net).

1. REPORT STRUCTURE

The purpose of this report is twofold: first to identify and review:

(i) language issues to be defined (Sect.2),

(ii) users and uses of a language specification (Sect.3), and

(iii) language specification techniques (Sect.4), and then

(iv) to identify (Sect.5) and review (Sect.6) the requirements which the above three aspects imply of the Ada FD specification.

Given the (current) state-of-the-art in formal language definition techniques, section 6 is a preview of the extent to which we, today, believe that the Ada FD will fulfil these requirements.

A final version of this report, to be edited when the project is (almost) completed, will attempt to assess whether these requirements have then been met.

So: we see the three subject categories (i-ii-iii): language issues, user groups/expectations and specification techniques as almost orthogonally (independently) setting the scene for our endeavour. Exactly how we see these subjects determining our task is then detailed in section 5 (iv).

The reader is therefore asked to regard sections 2-4 as independent approaches to the problem at hand: the construction of an acceptable definition of Ada.

The reason for listing so many language issues, uses, and users is the following: we wish the resulting specification to address as many of these as are relevant. Or, putting it in the opposite: not doing a proper analysis (viz. realizing which could be the potential language issues, uses and users) would most probably hamper our specification work. We are trying to avoid making a specification for its own sake.

We want a specification which is of relevance, which is important, and which, hopefully, is to be influential. The areas it could influence are those of the language issues, and the language uses and users.

2. LANGUAGE ISSUES

A language specification should take a clear stand on which language issues to cater for, and which to dispense with. Therefore, we list a number of language issues.

In sections 5 and 6 we shall then conclude which of these issues will be in the domain (ie. within the scope) of the specification.

A number of language issues, other than specification, can be identified. The meaning of the concept "language issue" should transpire from the below analysis. There is no guarantee, neither that this is a complete list, nor that it is a list of independent (orthogonal) issues. Since the subject of "language issues" itself is not exclusively a formal one, but also relates to pragmatic issues (such as the interests of individuals, groups, and institutions), and derives from their expectations, the treatment necessarily has to be informal. Yet, we shall try to be systematic.

We see the language issues to deal with:

(0) Language Design

(1) Language Properties
   -- Determinism, non-determinism, concurrency, incorrectness, erroneousness, undefinedness, implementation dependency, etc.

(2) Language Use
   -- Use in the programming situation, by the ordinary programmer, for the development of worthwhile programs
   -- Use for program documentation

(3) Language Implementation
   -- Development of interpreters
   -- Development of compilers
   -- Development of support tools: documentation aids, proof systems, etc.

(4) Standardization

(5) Education: teaching and training, textbooks and reference manuals

(6) Research

2.0 Language Design

The language has been, or has to be, designed.

In designing a programming language, the designer usually has two other concerns: programming techniques (methodology), and compiler (interpreter) implementation. The designer should, however, have a third concern: ease, or elegance, of explaining the semantics. Formal specification may offer a tool to be used actively by language designers.

One last concern could be: to what extent, in what sense, and how (if relevant) a language design permits language subsets or extensions.

(The current version of Ada is constantly undergoing re-design. It is not planned that the Ada FD project should offer explicit liason to the on-going ISO Ada LMC (Language Maintenance Committee). We shall, however, inform the ISO Ada LMC about problems arising from potentially questionable language design. But that is not an active design issue, such as "what effects do I get, if I design a construct such-and-such?". Our input to the ISO Ada LMC is more of the passive character: "since you have now designed this/that construct such-and-such, let us inform you of the following problems: ...".)

2.1 Language Properties

The language has properties.

The issue here (in the context of given, accepted, and reliable language designs) is: Independently of the detailed specific semantics, how can we characterize and classify language features

so that a design (and its specification) most "directly" and faithfully, abstractly defines these features.

The kinds of language construct properties we have in mind are:

(i) deterministic features, like statement sequencing and specific order of elaboration (e.g. left-to-right)

(ii) non-deterministic features, like arbitrary order of evaluation (e.g. subprogram parameters)

(iii) concurrency (parallelism), like tasking

(iv) incorrectness: certain syntactically correct composed features not being defined semantically

(v) erroneousness

(vi) undefinedness

(vii) implementation/target machine dependent features

The problem at hand is: for each construct, or combination of constructs to classify it according to the above categorization, and then, if feasible, to find and apply a most fitting definition technique.

2.2 Language Implementation

The language has to be implemented. Hence, implementability is a language issue.

Language processors are either:

- interpreters,
- compilers, or
- support tools

2.2.1 Interpreters

It is, for example, a language issue to which extent various bindings of a program (e.g. of its names to their meaning) can only be done at run-time. That is: how dynamic are these bindings in the sense of names being bound (in different runs of the program) to different kinds (or types) of objects. The more so, the more programs have to be interpreted.

2.2.2 Compilers

At the opposite end of the binding spectrum from all being fully interpreted, we have fully static bindings, i.e. bindings the validity of which can be checked before run-time, i.e. at so-called compile-time. The more so, the more programs can be compiled!

...

So the position in the spectrum from compilability to intrinsic forced interpretability is a language issue. It is a relevant question whether a language definition reflects this position in the spectrum.

2.2.3 Support Tools

A number of different kinds of support tools can be identified.

Programming-in-the-Small Tools
Program Re-use Tools
Programming-in-the-Large Tools
Program Verification: Theorem Prover and Checker Tools
Separate Compilation Supports
Program Linking and Loading Tools
Program Testing and Validation Tools
Program Debuggers
Program Execution (Run-Time) Supports
Program Maintenance and Version Control Tools

The extent to which a language lends itself, through distinct or similar facets, to each of these tooling and support possibilities (whether desirable, or relevant) is a language issue.

2.3 Language Use

The language is to be used.

The issue here (independently of a formal language specification) is: the use of the language in the programming and in the program documentation situations.

An additional language question may be: which are the various uses (the categories of applications) into which the language will come?

We attempt, without here expecting to be exhaustive, to list some uses:

Computation-Intensive:	Numerics	(Number "Crunching")
	Symbolics	(Algebraic Computations)
Process-Intensive:	Control	(Embedded systems)
	Communication	(Networks)
Data-Intensive:	Databases	(Input Systems,...) (Information Systems)
Deduction/Inference-Intensive:	AI	(Knowledge Based Expert Systems)

These various uses are made by users, and these users expect to find (in a language definition) answers to questions related to each of the above-listed areas.

2.3.1 Programming and Program Proofs

Programs have to be developed, and some of them proven correct.

Therefore, the issue is : in which ways does the programming language lend itself to, for example, stepwise, modularized (etc.) approaches to development, and to reasoning about worthwhile programs.

2.3.2 Program Documentation

Programs have to be documented.

The issue therefore is: through which mechanisms does the language lend itself to program-documentation.

2.4 Standardization

A language can be standardized.

The ease or difficulty with which (1) a language can be standardized, (2) a standard can be adhered to, and (3) a standard can be maintained is a language issue.

2.5 Teaching

A language has to be taught, i.e. it has to be understood.

The ease or difficulty with which a language can be taught and understood, and textbooks and reference manuals written, is a language issue.

2.6 Research

A language is a live or a dead object.

The excitement (disappointment) generated by a good (bad) language design is reflected back into the scientific community. The foundational and methodological research into a language is a language issue - even when this research is done for pragmatic, opportunistic reasons.

2.7 Conclusion

We have listed some language issues. We have tried not to commit ourselves, or the parties involved in these issues, yet to any stand on these issues vis-a-vis a formal definition, and which of these issues an Ada FD reflects! The next section will take a first view of this latter concern.

3. USERS AND USES OF AN Ada FD

Various authors have listed categories of users.

In [2] we find:

> Users, implementors, and textbook writers.

In [4] we find:

> Users, educators, manufacturers, compiler writers, and theorists.

In [5] we find:

> Designers, implementors, and programmers.

In [6], we find the best list so far, including:

> Novice/practising/sophisticated progammers, local experts, educators, implementors, validators, designers and language reviewers, standards people, programming methodologists, and formalists.

These lists of users imply similar lists of uses. Below, we have basically followed the proposal of [6].

Let us assume that a perfect, all-encompassing formal definition of Ada, with all the desirable properties (whatever they are), could be produced! By whom and to what would or could such a definition potentially come into use? This section tries, on the background of the tentative enumeration of section 2, to list such potential.

Some [4] say that "a language definition should be the ultimate authority on a language", and "it must contain answers to all questions about the language". [4] does not outline the nature of these questions. Our section 3 is an attempt to do so.

3.1 Language Designers [5]

Usually, language designers are experts in program coding (i.e. program implementation), and in language implementation (typically "compiler writing"). Language designers do not, with rare exceptions [27], master natural (national, e.g. English) language stylistically well. At least not to the degree that is really needed for writing a precise reference manual. Despite this, language designers are most often the only, or at least the first, to write such an informal document. Language designers are to a somewhat larger degree capable of reading the now classical formal definition styles ([37]).

Despite the above, a rôle of a formal definition is to advice the designer of all language trouble spots, i.e. ambiguities, undefinednesses, inconsistencies, and incompletenesses.

Another rôle of a formal definition derives from the process of attempting to formally define a language. The ease (or difficulty) with which this definition process proceeds could be an indication of some "measure" of naturalness ("artificiality") of the proposed language construct. [This last postulate is not objective in cases where the chosen definition method (technique and semantic language) is ill-suited for its purpose, anyway].

Section 5 will state the current Ada FD position on the above points.

In summary, we conclude that ([6]) "language designers (and distinguished reviewers) should be primary users of an Ada FD - also in their rôle of advising standardizing committees about language changes".

3.2 Implementors [5]

In [5], three kinds of expectations that implementors might have of an Ada FD are identified:

(1) "advice concerning the meaning of some language feature", incl. "what it is supposed to do",

(2) "advice concerning implementation", and

(3) advice concerning "actual certification of compilers, or possibly compiler components".

[In [5] the above (1-2-3) are stated w.r.t the functions of a validation centre - rather than, as here, w.r.t an Ada FD.]

Certainly an Ada FD should resolve (1).

Insofar as an Ada FD is constructively defined, e.g. in a model-oriented denotational or operational semantics formalism, such an Ada FD could also give some kind of advice concerning pt. (2).

And insofar as an Ada FD can serve as a reference point w.r.t. validations, it can also (circularly) satisfy point (3).
Ways of serving as a reference point for certification (validation, etc.) are: (I) implementations could be proven correct w.r.t an Ada FD, and (II) implementations could be subject to testing by means of a set of correct and incorrect programs automatically generated from an Ada FD.

Section 5 will state the current Ada FD project position on the above points.

In summary, we conclude that ([6]) "implementors of compilers, interpreters, and support tools (interfacing to the syntax and semantics of Ada) need the Ada FD to decide on language issues".

3.3 Programmers

Following [6] we sub-divide this group into:

"(a) Novice Programmers:
-- e.g. having never heard of generics,

(b) Practising Programmers:
-- e.g. users of generic packages,
(c) Sophisticated Programmers:
-- e.g. producers of generic packages."

The position of [6] seems to be that neither of these groups should or will be potential users of an Ada FD. We tend to concur.

Instead, we believe that other user groups, in particular educators (writers of reference manuals and textbooks on Ada), and local experts (i.e. programming consultants), should/will act as intermediaries between programmers and an Ada FD.

In [5], on the other hand, a useful emphasis is put on the rôle of the programmers vis-a-vis an Ada FD: discrepancies (found by programmers)

(1) between a validated compiler and reference manuals or

(2) between two validated compilers, and

(3) clarifications of language points which are unclear in reference manuals

should be duly communicated to the definers and, subsequently the maintainers of an Ada FD.

3.4 Standardization

Members of language standardization committees (ISO, ECMA, ANSI, BSI, DoD) and language maintenance committee (Ada LMC) have many rôles [6]: "they act upon advice from validators (to resolve mis-interpretations), from designers and reviewers (to decide (between) possible changes)", and from implementors (e.g. to help easing the burden of compiler realization); and they are, in cases, otherwise influenced by e.g. manufacturers' wishes (to sub- or super-set the language, to interface it to database languages, etc.).

It is believed that members of such committees [6] "should be most familiar with an Ada FD, and interested in its maintenance". Their use of an Ada FD and its updates should be to help decide on, or between proposed language changes. In attempting to introduce a language change into an Ada FD, insight might be gained as to the desirability of such a change. We are referring here to the ease (naturalness) (or difficulty (artificiality)) with which such a change can be introduced into an Ada FD, to the containment (or propagation) of language changes, and to the reduction (or expansion) in size of an Ada FD that proposed language changes might incur.

3.5 Teachers, Instructors and Programming Consultants

To this class we count the writers of text books, reference manuals, and programmers' guides on Ada. And we shall illustratively see their rôle vis-a-vis Ada in this light, only.

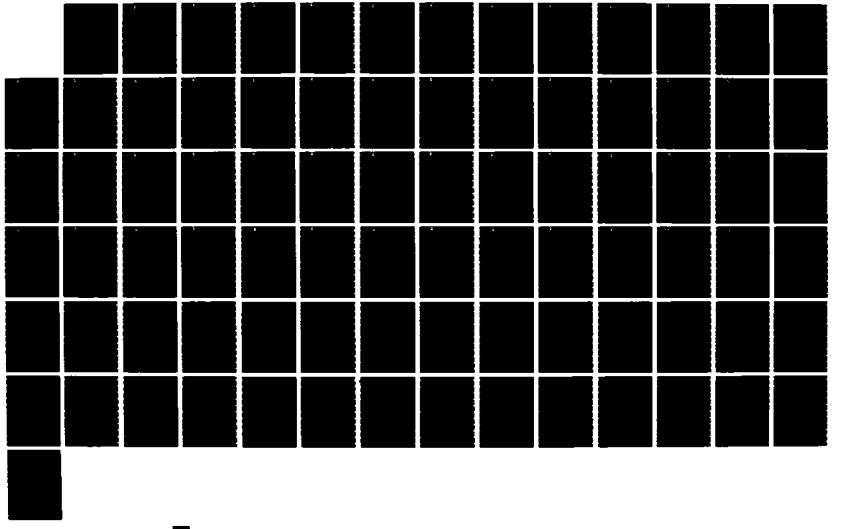
It is [6] "expected from them that they spend some time studying an Ada FD". And it is believed that they should be able, from such reading, to extract various levels of informal documents, also representing various views on Ada. Included among these, should be the ability to extract various kinds of language subsets for student and programmer introduction and programming specialization.

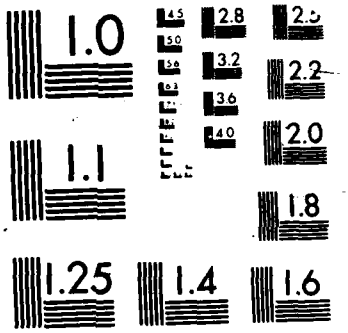
In addition, they should be able to consult the Ada FD on language issues arising from their involvement in deeper technicalities, e.g. where, on behalf of programmers and implementors, they find the kind of discrepancies listed in section 3.3

3.6 Scientists

In [6], this group of users of an Ada FD is also called formalists.

AD-A172 747 PROCEEDINGS OF THE IDA (INSTITUTE FOR DEFENSE ANALYSES) 4/4
WORKSHOP ON FORMA..(U) INSTITUTE FOR DEFENSE ANALYSES
ALEXANDRIA VA W T MAYFIELD ET AL. NOV 85 IDA-M-135
UNCLASSIFIED DECL IDAH030579 IDA/HQ-85-30579 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0

(It will basically be computer scientists who will produce an Ada FD.)

Formalists (computer scientists) could potentially be extensive users of an Ada FD. Their uses of an Ada Fd could be as a basis for [6]:

(1) the derivation of proof rules for Ada programs - given that the Ada FD in question is not itself formulated in terms of proof rules;

(2) the derivation of Ada program transformation rules - possibly for use in programming or in compiler optimization; and

(3) the derivation of a co-ordinated formal semantics of a specification language for defining program properties.

Point (3) is an extension of points (1-2). In addition, an Ada FD could be used by formalists as a departure point for:

(4) investigations into its semantic foundations, other than (1-2-3), for example into areas that may not explicitly be covered by an Ada FD (areas such as: fairness, performance, complexity, realtime concerns, etc.); or

(5) investigations into other, competitive semantic definition methods for the sake of fruitful (counter-)arguments, the further progress of science, etc.

The production of an Ada FD is a considerable undertaking and will result in a very large document. Such a document will likely not be perfect - solving all recognizable issues, let alone identifying all such. An Ada FD is therefore expected to be a live document continually being questioned by scientists.

3.7 Validators

The issue of the relationship between an Ada FD and Validation is treated in [3] and [5-6].

[5] examines the rôle of an Ada FD document w.r.t. the functions of an Ada compiler validation centre. [3] examines postulated desirable relations between an Ada FD and the so-called ACVC test suite. [6] effectively summarizes [5].

The issues raised by [3] are dealt with in our section 5.10. We now summarize [5-6] and also inject additional points.

By validation in general we mean a process, between a customer and a supplier, whose aim it is to improve confidence in the correctness of a specification, a design, or an implementation (i.e. code), or in the claim that a specification, a design, or an implementation fulfils given requirements.

In general we see such a process as being carried out by some combination of formal proofs of correctness, and test case execution (i.e. testing). A proof of correctness would be of an implementation with respect to a specification. The proof (of a theorem) could either be provided by a theorem prover, or a manually provided proof could be verified by a proof checker, or by some combination of the two. The theorem is stated by supplier and customer in unison. The selection of test cases and their expected results is likewise a contractual issue between supplier and customer.

Testing is a combination of two things: (1) a systematic and organized search for a counter-example to a claim that a specification, a design, or an implementation is correct, and (2) the (possibly partial) execution of a specification (etc.) in order to demonstrate that it (they) fulfils some non-functional requirements. Correctness proofs usually only tackle functional (formalizable) requirements. On this very general background, the validators' use of an Ada FD is many-fold:

(1) as a basis for organizing the systematic search for counter-examples to claimed proofs,

(2) as a basis for generating test programs - both correct and in-correct, and

(3) as a basis for deciding on tests for non-functional, i.e. unspecified properties.

Concerning (2) there are two issues related to the possibility of using an Ada FD: (I) to synthectically generate correct and, desirably, incorrect Ada programs, and (II) to generate the kind of answers a compiler should output upon (or while) executing such generated programs.

Very little knowledge is available in these areas (1-2-3). [6] mentions [28] as a possible source of inspiration.

4. LANGUAGE SPECIFICATIONS

References [10,11,12,15,16,17] contain specific language specification proposals. [5] actually lists a more refined list of specification varieties than [16]. Section 4.0 borrows from [5]. Section 4.2 borrows from [16].

4.0 Language Description Categories

A language can be defined in either of a number of alternative, contrasting or complementing ways:

(1) through a Reference Manual [23], and its Rationale [24]
(2) through an Implementation Guide [20]
(3) through an Implementation [19]
(4) mathematically

[5] also lists the possibility of a pseudo-formal (notational) description, which lies somewhere between (1) and (4) in that only an informal definition might be given for the description language itself, whereas its syntax looks formal. In a sense, [21], [25], and, to a small extent, [22] could be rightfully accused of being pseudo-formal.

4.0.1 Reference Manuals and Rationale

By a language reference manual (LRM), we operationally understand an informal document which in a technically carefully controlled dialect of a natural language, e.g. English, explains the semantics of another language, namely the programming language. (The syntax is usually given by some BNF-like grammer.)

By a programming language rationale is understood a necessarily informal document which explains the pragmatics of the language. (It is informal since pragmatics is an informal issue.)

An LRM [23] does exist, and a draft partial Rationale [24] now exists.

Usually, LRMs suffer from lack of precision due to the use of a natural (national) language. To alleviate the lack of precision, the description often becomes stilted, legalistic. [26] could be accused of that.

Rationale documents are the source of the non-functional definitions and pragmatic information - where LRMs tend to concentrate on syntax and functional semantics.

LRMs and Rationales are deemed indispensable for reasons of readability, but suffer in accesibility and referenceability, as defined in section 5.6.

4.0.2 Implementors' Guide

By an implementors' guide (for some programming language), is understood a (formal or informal) document which lists any number of hints on how to implement a processor for that language.

For Ada, there was an implementors guide [20]. Its usefulness was rather limited. [20] suffers from four things: (1) it is based on an informal LRM, (2) it was issued at a time when Ada was still being (re-)designed, (3) it is itself informal, and (4) it could be critisized for reflecting an outdated compiler writing technology.

As a reference to a programming language for others than implementors, an implementors' guide is usually almost useless.

Insofar, as an implementors' guide takes the opportunity to clarify language semantics that is left unclear in an LRM, such a guide is useful, but we consider the place and time ill-chosen precise semantics should be given in the LRM and in an Ada FD.

Insofar, as an implementors' guide enumerates ranges of permissible implementation choices, such a guide is considered most useful.

Finally, the concurrent existence of both an LRM, an Implementors Guide, and possibly an Ada FD poses the problem of maintaining consistency. Especially, if the last two are derived from an LRM. We advise the other way around: the derivation of an LRM and Implementors Guide(s) from an FD.

4.0.3 Compiler as Language Describer

In the 1960's it was a commonly taken view that compilers defined their languages [19]. As long as programs in what was believed to be one language were not ported between different compilers (usually on different computer mainframes), no real harm seemed imminent. With porting, or copying program fragments between different installations, problems became apparent. By porting compilers, these problems seemed to disappear for a while. There was, and maybe still is, a need, within one mainframe to make use of distinct processors for supposedly the same language, e.g. compilers which optimize, for production run-time performance, or interpreters with good programming time debugging facilities, or, perhaps more relevant, which 'prove' program properties!

We take the view that neither of these kinds of processors define their language, but that they are "derived" from a (possibly formal) definition.

4.0.4 Formal Definition

The fourth language description, or definition, possibility is that of a formal definition. We devote sections 4.1-2 to that subject.

4.1 What is meant by 'Formal'

From the terminology, appendix B, we get definitions of what is meant by formal: formal development, formal document, formal language, formal method, and formal proof. The essence of 'formal' is that whatever is formal is expressed within a formal system, i.e. in a formal language, either being, or accompanied by, a proof system.

The notion of a formal system is invented in this century. It was introduced in order to tackle the foundations of mathematics. As such, 'formal systems' belong to meta-mathematics. And as such, they certainly run the danger of loosing a hold in reality [40].

Mathematics, for milleniums, was tightly rooted in observations in physics and in everyday human life. Accordingly, much mathematics was presented with analogies to this reality. Meta-mathematics tends to be presented at most by reference to mathematics - a universe in which arbitrary, finite and infinite, imaginable and un-imaginable objects may exist.

Computer science deals, not with mathematics, but with the objects that may exist in machines and in their man-made creation. To do computer science, we use mathematics. But we follow, in this project, the dogma that this mathematics should be firmly related to the programming language world as outlined in sections 2 and 3.

4.2 Formalization Techniques

4.2.0 Deductive and Model-Oriented Specifications

For the purposes of the present subject, an Ada FD, we distinguish between two styles, or aims, of formalization:

> Deductive, Assertional, or Property-Oriented, and
> Constructive, or Model-Oriented.

(We refer to the terminology for the definition of these terms.)

Proof system oriented specifications, i.e. specifications which directly lend themself to reasoning (about the object defined, and to be implemented), are typically deductive (assertional, property-oriented). Such specifications, when expressed freely, usually require a proof of their (own) consistency and completeness, and of the fact that they do define something. That is: that they have at least one model.

Model-Oriented specifications, as the name implies, directly describe, i.e. are, the models. It is in that sense that they are constructive.

Usually, one desires the properties, but specifies a model. Several reasons may account for this: (1) most software people, today, are trained to think model-oriented, (2) model-oriented specification techniques are, today, capable of tackling the definition of far more complex systems than the deductive techniques appear to be, (3) what is specified has to be implemented, i.e. one has to find a model - sooner or later, and (4) the state-of-the-art in going from a deductive definition to a constructive specification is somewhat lacking.

A property-oriented definition lies close to the customer's way of formulating his requirements, whereas a constructive specification similarly lies close to the supplier's way of thinking of his job: that of developing an implementation from the specification.

Ideally, we would like to see first a pure, deductive definition (of, say, Ada), and then, from it, rigorously derive a constructive specification.

Realistically, we may hope that it is possible to prove what is deductively defined (i.e. a deductive definitions' axioms (etc.)) to be satisfied (i.e. to be properties) of a construc-

tively specified model.

From the above, the reader may guess that the current Ada FD project takes its departure point in a model-oriented world, but that everything will be done, within the evolving Ada FD, to secure the possibility of deriving properties rather directly.

A number of specification techniques cover the span from deductive to model-oriented definitions: Axiomatic, Algebraic, Structural Operational, Denotational and Mechanical Semantics. A short, very cursory survey of these will now be made.

4.2.1 Axiomatic Semantics

Roughly, an axiomatic semantics specifies relations between states of the specified system.

In an axiomatic specification of a programming language, its semantics is given in terms of axioms and deduction rules (for using these axioms).

[36, 37] are seminal references on this subject.

Such axiom systems seem ideal as proof systems for the language they specify. The problem is, however, that they become rather cumbersome, if not outright in-applicable, when having to deal with a complex language like Ada. Focal points for complexity are: gotos, procedures, parameter passing, and tasking.

This rather negatively sounding dismissal of Axiomatic Semantics as a basis for an Ada FD must not be mis-understood.

Beautiful languages can be designed and effectively used, their semantics being so specified. [42, 43] provide convincing evidence. Here, the axioms are expressed in a different style and are called laws. Most likely, future languages will be designed on the basis of their proof system

being simultaneously evolved!

4.2.2 Algebraic Semantics

An algebraic semantics specifies the meaning of a system as a class of algebras.

In an algebraic specification of a programming language, its semantics is given in terms of an algebra presentation, consisting of a signature and a set of axioms. Usually, an algebraic presentation is (syntactically) constrained so as to guarantee the existence of models. The meaning of an algebra presentation is usually some class (or category) of algebras. The axioms are usually equationally specified.

[35] provides today's most accessible introduction to algebraic semantics.

Again, we find that algebraic semantics specifications ought to be ideal as a basis for language proof systems. No algebraic specification has yet been given for any sizable classical language (ALGOL 60 or larger), let alone for concurrency aspects of such a language. Problems in their applicability (in addition to those of axiomatic semantics techniques) seem to be their inadequacy in handling higher order functions (procedures with procedure parameters) and tasking.

This rather negatively sounding dismissal of Algebraic Semantics must not be misunderstood. What we are indicating is only that we may not be defining (parts of) Ada directly in terms of algebraic semantics. You may find, however, that the definition style we eventually adopt will involve (a) definition language(s) the semantics of which may be given algebraically.

4.2.3 Denotational Semantics

A Denotational Semantics defines the meaning of a system to be a set of mathematical objects (like sets, functions, categories).

In a denotational specification of a programming language, its semantics is usually given as follows. First, one identifies the specific mathematical object one wishes to attach to simple identifiers of programs. [Examples are: variable identifiers may denote functions from so-called environments to locations, label identifiers may denote so-called continuations, i.e. functions from stores to stores, procedure identifiers may denote functions from argument (denotation) lists to continuations, etc.] Thus, we first establish the meaning of simple language constructs. Then we express the meaning of composite language constructs as functions of the meaning of their constituent components. (This latter is really an algebraic (homomorphic) principle, and not necessarily characteristic only of denotational specifications.)

Denotational specifications directly specify models. As such, they are not directly useful as proof systems, and not much systematic work has been done, nor are systematic techniques available for the extraction of proof systems from denotational specifications. The power of denotational semantics is that it deals effectively with gotos, procedures, parameters, and exceptions and with most other deterministic language features. Problems of denotational semantics are shared variables, Ada-like processes, and non-determinism. We refer here to the availability of techniques proven on large scale applications. There are recent research results (like [44]) which appear very promising, but for this project they have, unfortunately as it may seem, to be discounted for exactly the reason of their experimental nature.

4.2.4 Structural Operational Semantics

A Structured Operational Semantics (SOS) defines the meaning of a system by the set of all allowable transition sequences that may be observed in a system (state) while subjected to execution.

It is in this latter sense (execution) that SOS is "operational". It is structured in that transition rules are inductively specified, based on the structure of the system input language.

[32, 45] provide first and latest references to SOS.

An SOS specification is usually given in terms of a set of transition rules and rules of induction for using the former. A transition rule consists of a triple: the "before", the "after", and the "condition" (label) under which a system may transit from a before configuration to an after configuration. Configurations and labels are rather free-wheeling notions, and may involve state components such as stores, program fragments and other control information.

SOS specifications eminently model non-deterministic and concurrent language aspects, in addition to trivially being able to model deterministic features. SOS specifications appear promising as a basis for direct or derived proof systems. SOS specification techniques, when brought to bear on the full complexities of Ada tend to result in rather complex configurations and labels.

4.2.5 Other Specification Techniques

We have indicated that none of the above techniques, except perhaps SOS, is fully capable of handling the specification of all aspects of Ada.

In addition to the above techniques, others have been used and/or proposed:

SEMANOL [17, 46], VDL [34], Meta-IV/CSP [21]. This is not the place for even a cursory description of these more operational (mechanical) definition styles.

It is implied in the above rather cursory remarks that the present Ada FD will not entirely rely on any one of them. [17] points out, very importantly so, the need for, first, establishing a model for the underlying semantics when dealing with a complicated system like Ada. That is: that one, in a sense, starts afresh, forgetting, for a while, the dogmas of e.g. Axiomatic, Algebraic, or Denotational Semantics, i.e. of their underlying mathematics.

4.3 The Ada FD Approach

Although not intrinsic to the purpose of this document, we do present a very cursory overview of the approach to an Ada FD currently taken within the project.

D-SMoLCS [33]

The Method

In the D-SMoLCS (Denotational SMoLCS) approach, the formal semantics of Ada is presented in two hierarchical, top-down steps:

(I) Denotational Model

(II) Semantic Algebras (SMoLCS)

It is developed in the reverse order of these!

-- I: Denotational Model

The denotational model, in order to model all aspects of concurrency and non-determinism, will be expressed using a number of operators like e.g. | (for "in parallel"), (overloaded) + (for "choice"), (for "followed by"), etc. This model can be based on the use of the exit mechanism, and in either an imperative or an applicative style (as possible in Meta-IV), or on the use of a continuation mechanism (also possible in Meta-IV).

(As a consequence, the resulting model is one which can be read by humans.)

-- II: Semantic Algebras (SMoLCS)

The denotations of the model presented in the first step are presented in this step. On one hand there are these denotations, and, on the other hand, there are operations (like "|", "+", " ", ",", ";", etc.) on them. The meanings of these operators are likewise presented.

These presentations are given in five configurational, bottom-up sub-steps:

Basic Transition System

In the Basic Transition System sub-step, we specify what the element Actions of the individual processes are.

Synchronization

To a Basic Transition System, we add rules (parameters) governing the synchronization points between processes, i.e. we define atomic actions.

Parallelism

Given a Synchronized System, we add rules (parameters) for the (parallel) (e.g. mutually exclusive shared update) composition of processes.

Monitoring

Given a Parallel System, we add rules (parameters) which define restrictions on the behaviour of the processes of the parallel system.

Observational Semantics

Given a Monitored System we may now wish to interpret the given semantics at any one of a number of levels of observational abstraction: input/output, interleaing, fair-merging, truly parallel, etc. This is done by suitably parameterizing the algebraic specifications which have been given of the synchronization, paralleism, and monitoring operators.

D-SMoLCS: Its Semantics

SMoLCS can be embedded in an SOS specification. For the sake of obtaining the much desired properties of the Observational Semantics, Algebraic embeddings have instead been used. Other, more functional approaches are conceivable.

The Current Ada FD Components

Basically, we plan to divide the Ada FD itself, into two-by-three components:

(1) A static, and
(2) a dynamic semantics specification - each consisting of

(1) syntactic and
(2) semantic domain specifications, and
(3) the semantic function definitions/equations.

The domain specifications will be specified in Scott theory, i.e. as possibly reflexive domains. The notational style is basically that of Meta-IV.

The static semantics functions will basically be centered around a pure, applicative Meta-IV subset (e.g. not using the exit mechanism), but may define certain static semantics domains and operations upon their objects algebraically.

The dynamic semantics is presently planned to be based on the D-SMoLCS approach. For practical reasons it will consist of three parts: "sequential Ada", "tasking Ada", and "Input/Output Ada".

5. REQUIREMENTS TO THE Ada FD

First, we give some overviews. The Ada FD, it is claimed in [1], should be:

(1) A Legal Contract,
(2) Consistent and Complete,
(3) Comprehensible and Precise,
(4) Correct, and believed Correct (latter from [8]),
(5) Accessible and Referenceable (from [8]),
(6) Permissive - where appropriate (from [8]),
(7) Un-biased,
(8) Suitable as a basis for:
    (8.1) writing user manuals, textbooks, and primers,
    (8.2) developing language processors, and
    (8.3) validation, - and
(9) Suitable as a basis for:
    (9.1) proving correctness of Ada processors,
    (9.2) proving correctness of specified Ada programs, and
    (9.3) generating test-programs for validation.

In [2], we find that an Ada FD might:

    (1) resolve ambiguous points in the existing standard,
    (2) omit points of the standard, and
    (3) include points not addressed in standard.

(It should, to be proper, be observed that [2] asks "to what extent an Ada FD" should address (1-2-3).)

In [3], it is argued that an Ada FD should somehow be correlated to the so-called "ACVC test suite" (jargon for an "Ada Compiler Validation Capability" collection of some 2000 test programs).

From [4], we extract the following requirements:

    (1) Basis for Compiler Writing,
    (2) Compiler Validation,
    (3) Proof of Compiler Correctness,

(4) Basis for deriving an Axiomatic Proof System,
(5) Basis for deriving Program Transformation and Optimization Rules,
(6) Basis for Rapid Prototyping,
(7) That it be Unique (no Alternatives),
(8) Complementing and Consistent with LRM,
(9) Exposing existing/current LRM inconsistencies,
(10) Guiding Language Clarification,
(11) Machine Processable, and
(12) Correlated to ANNA [18].

In [5], we find the following requirements:

(1) Completeness and Consistency,
(2) No Ambiguities,
(3) No Over-Specification,
(4) Readable,
(5) Maintainable,
(6) Modular, and
(7) Basis for Proofs.

In [6], related to user categories, see sect. 3 above, we again find a good overview of requirements to an Ada FD: namely that it be suitable for use:

(1) as a source document: by formalists (i.e. be formal), validators (i.e. be executable), implementors, educators, and designers,

(2) as a canonical contract (standards) document: by standardizers,

(3) in implementations: free from implementation bias and permissive where appropriate,

(4) as a reference document: by educators and local experts, and be: comprehensible, concise, accessible and referenceable, and

(5) by formalists, validators, and implementors: consistent and complete, and believed correct.

[6] seems to have used [1] and [8]'s requirements, but relating them to user groups which are missing in [1] and [8].

In [7], we find the requirements that an Ada FD should be:

(1) machine readable (for tool development.),
(2) human readable, and
(3) accompanied by user's guides, structured by user target groups, (e.g. as listed in section 3 [5]).

From these lists, we have then extracted the structure of this section:

1 Legal Contract
2 Consistent and Complete
3 Comprehensive and Concise
4 Correct and Believed Correct
5 Accessible and Referenceable
6 Permissive
7 Implementation Independent
8 Basis for Processor Development
9 Basis for Validation
10 Basis for Proof Systems
11 Mechanizable
12 Basis for Rapid Prototyping
13 Correlatable
14 Basis for Document Derivation
15 Maintainable

5.1 Legal Contract

By an Ada FD constituting a 'legal contract', we understand something that eventually borders upon the legal meaning of 'legal contract', namely that a user can rely on his 'formal' understanding to be the same as the developers' similar understanding. This point is then ultimately seen as leading to the derived requirement that a definition is formal.

Thus, a formal definition could ultimately serve as a legal document in a court of law.

5.2 Consistent and Complete

By an Ada FD being 'consistent and complete', we mean what these terms mean in mathematical logic.

5.3 Comprehensive and Concise

By an Ada FD being 'comprehensive', is meant a relative thing: that any person, brought up in reading formal definitions (of such-and-such style), will have no undue, or unreasonable difficulty in reading such definitions.

By an Ada FD being 'concise', we similarly mean a relative thing: that the definition is precise and not unduly long.

(Both properties are definition-style independent, i.e. are solely a function of the success with which the definers have achieved their goal, i.e. their ability to use a given definition style according to its best intentions.)

5.4 Correct and Believed Correct

A formal definition of the functional aspects of a language is 'correct' if it meets the requirements of the customer of that language. Mostly, these requirements are informally stated. Hence, we mean something not achievable when we require a definition to be correct! Or we could claim that a definition, if it is complete and consistent, "by definition" is correct.

The phrase, 'believed correct', is therefore introduced. By an Ada FD being 'believed correct', we mean something relative: if the definition is formal, then it "mirrors", in some informal sense of "equivalence", a "similarly official" informal defini-

tion. That is, there are no obvious discrepancies between the Ada FD and the original intentions (as for example expressed in language requirements documents), or informally expressed specifications, viz.: the LRM.

5.5 Accessible and Referenceable

By an Ada FD being 'accessible and referencable', we mean something a bit more absolute.

To the Ada programmer and the Ada language processor developer, the Ada language consists of a number of commonly agreed, verbally identifyable, semantic ideas (concepts, constructs, notions).

Answers to questions about properties of each of these should preferably be found, say, within the short span of a page of a definition, i.e. be accessed by a rather direct look-up process. Vice-versa: once this is the case, then one can refer uniquely to such definition pages (etc.).

(This 'ability' is partly a language, and partly a 'definition' property: i.e. if the language otherwise permits, then the above 'locality' property should be satisfied.)

5.6 Permissive

By an Ada FD being 'permissive', we mean that the definition, ideally, expresses all permitted aspects of 'order of evaluation', 'optimization', 'parallellism', 'non-determinism', etc.

5.7 Implementation Independent

By an Ada FD being 'free from implementation bias', we mean something similar to permissiveness: namely, that the definition does not unduly favour one style of implementation over another where such choices were not intended by the language architechts.

5.8 Basis for Processor Development

By an Ada FD being 'suitable as a basis for interpreter, compiler, and support system development', we mean that the development covers each and every aspect of the language, and that this coverage can be secured through use of the definition.

-- Correctness of Processor Development

By an Ada FD being 'suitable as a basis for proving correctness of Ada processors', we only mean the constructive, a priori, proof of correctness of the development of an Ada processor (not any, a posteriori, given such processor). Thus, we are, here, only thinking of an Ada FD serving as the departure point for actual processor development. And we primarily think of this development as transforming, refining and enriching an Ada FD, via stages of development, ie. via abstract and concrete designs, to actual implementations (code).

5.9 Basis for Validation

By 'suitable as a basis for validation', we mean: it should be possible to construct a test-suite of Ada programs for the purposes of testing any given compiler.

The derivation of test programs should be transparent: i.e. a human should be convinced that these are "real" test-programs. The derivation should be "exhaustive", i.e. convincingly span a necessary spectrum of programs. The derivation should foresee a range of implementations: i.e. the test programs should not only test the 'language structure', but also foreseeable 'processor structures'.

-- Conformance with ACVC Test Suite [3]

By 'conformance between an Ada FD and the ACVC test suite', is meant one of the following:

(1) Ignore this conformance and "verify" conformance between LRM and an Ada FD.

(2) Use informal reasoning to argue that the ACVC test programs are indeed processed correctly by the Ada FD, or, which may be a possibility, show ACVC test programs to not be correct test programs (i.e. testing contrary to the intentions of the language designers, etc.).

(3) Use formal proof techniques, manually constructed and verified, to guarantee what (2) sets out to achieve.

(4) Use formal proof techniques, mechanically verified, to achieve (2).

(5) Use prototyping techniques (see sect. 5.12) to achieve (2).

(6-7) Use formal compiler derivation techniques (see sect. 5.8) to achieve (2).

(8) Execute the Ada FD itself, directly, to achieve (2).

5.10 Basis for Proof Systems

When saying that an Ada FD could be 'suitable as a basis for proving correctness of specified Ada programs', we assume that the Ada programs are somehow specified, i.e. that certain assertions are made regarding their properties, and then that the definition 'formally' permits these to be shown to hold.

This either assumes that the definition is expressed in the form of a (set of) proof system(s), or requires that a proof system (or set of proof systems) is derivable from the definition.

5.11 Mechanizable

By an Ada FD being 'mechanizable', we mean that it be machine processable.

Reasons for wanting 'mechanizability' are many-fold:

(1) It would facilitate maintenance, and, therefore that an Ada FD is kept up-to-date. See sect. 5.15.

(2) It would facilitate correlation to other mechanized Ada documents: the LRM, ANNA, DIANA, etc. See sect. 5.13.

(3) It could facilitate scientific experiments in the areas suggested in sect. 3.6.

5.12 Basis for Prototyping

The act of Ada prototyping leads to a prototype Ada compiler or translator, and typically involves transliterating, if possible, the Ada FD directly into some high level executable code, for example SETL [30]. Rapid prototyping means the speedy, inexpensive production of a piece of software that is acceptable as a vehicle for a number of customer "testing" purposes.

5.13 Correlatable

By an Ada FD being 'correlatable', we mean the systematic, exhaustive, and unambiguous mapping of the formulae of an Ada FD to the Ada LRM.

5.14 Basis for Document Derivation

By an Ada FD being 'suitable as a basis for writing user language reference manuals' (etc.), we mean that such informal texts should be reasonably easy to develop systematically from the Ada FD, and should be easy to relate back to the definition, e.g. so that their 'completeness and consistency' can likewise be asserted. Here, we are thinking of a wide variety of informal documents: reference manuals for, naive, mature, and sophisticated programmers, respectively, for implementors, for validators, etc.

5.15 Maintainable

By an Ada FD being maintainable, we mean that the entire FD be computerized in such a way that a number of tools can be developed for browsing through the FD, for following correlations between the FD and the LRM (and possibly the Rationale), etc. The MENTOR [45] system appears to be a good candidate for support for developing Ada FD maintenance tools.

5.16 Assumptions

The above, more-or-less direct requirements, are based on a number of assumptions [16]:

(A1) "The Ada language is 'complete and consistent'"

> To the extent that work on a formal definition of Ada shows this not to be the case, an arbitration procedure could be established to secure 'completeness and consistency'.
>
> The position of the current Ada FD project is this: if an inconsistency or incompleteness is properly identified, then no formal definition will be given, i.e. the discrepancy in question is left undefined!

(A2) "There are Formal Definition Techniques that will satisfy all of the above Requirements."
To the extent that this is not found to be the case (which we can immediately assume (!)), an arbitration procedure must be established for deciding upon acceptable compromises in definition style and/or in the use of composite, overlapping definition alternatives.

The position of the current Ada FD project is this: the trial definition project sub-phase together with its external reviews will constitute such a procedure.

5.17 Derived Requirements

The above (more-or-less direct) requirements and assumptions imply a number of derived requirements.

(D1) 'Complete and Consistent' implies "Absence of Ambiguities".

(D2) 'Permissive' and 'Free from Implementation Bias' implies "No Over-Specification".

(D3) 'Accessible and Referenceable' implies "Maintainable", and "Modular".

(D4) 'Suitable as a Basis for Implementation' implies that one can derive "Implementation Guide-lines" from the definition.

(D5) 'Formally Defined' implies the possibility of "Language Definition Tools" such as rapid prototypers for language (subsets), definition ('consistency and completeness') checkers, etc.

6. THE ROLE AND SCOPE OF THE Ada FD

This chapter summarizes the position of the current project w.r.t. the idealized requirements listed in sections 5.1-5.15 inclusive.

(1) Legal Contract: time is probably not yet ripe for the computing community (suppliers and consumers) to rely on and to trust an Ada FD to constitute a legally binding contract in a court of law.

To the extent that we satisfy all subsequent requirements, one may hope to see the present (legality) requirement being achieved.

Thus, we conclude that our aim is to fulfil this requirement.

(2) Consistent and Complete: we most emphatically desire to achieve this requirement. A stumbling block may, however, be the (pragmatic) "interpretation" we attach to various notions of erroneous, undefined, 'pragma shared', 'permissive' (e.g. in case of non-deterministic features), etc. - as opposed to the similar "interpretation" of our audience: the Ada Language designers, standardizers, other formalists, validators, etc.

(3) Comprehensible and Concise: again, we most emphatically desire our FD to satisfy these requirements. One of our approaches to achieving this is to correlate the FD minutely to the LRM; another is to annotate it, likewise minutely, to also give an English language rendition of the formulae. This process of achieving such acceptable correlations and annotations is expected to feed back to the formula presentation itself.

(4) Correct and Believed Correct: we wish to achive also this goal, and basically through the same means as mentioned in (3).

(5) Accessible and Referenceable: also this requirement is of utmost concern to us. We most definitely wish to achieve also this goal.

(6) Permissive: insofar as we are able to identify all such language properties (as have permissive, implementation-wise "non-deterministic" properties), we shall be expected to also have the Ada FD be permissive!

(7) Implementation Independence: the Ada FD will most likely be model-oriented. Denotational (i.e. model-oriented) definitions do not necessarily bias some implementation choices over others. The current Ada FD will, similarly to what was indicated under point (6) (permissiveness), strive to exhibit implementation independence.

(8) Procesor Development: the current Ada FD project, basically having its root in a model-oriented, but abstract way of defining Ada, will strive to produce a definition which can serve as a departure point for interpreter, compiler and other Ada tool development.

For classical denotational and operational definitions, like the DDC formal description of Ada [e.g. 21], well-known methods exist [47] which allow the systematic to rigorous development of compilers from the language definitions. The current project, although claiming that it will strive to produce an Ada FD which should serve as a basis for processor development, will, however, not address the specific issues of how to formally derive such processors from the currently contemplated Ada FD. How, then, do we justify our claim ? By reference to the model-oriented, yet abstract nature of the planned Ada FD, and by reference to e.g. [47]!

We believe that the planned Ada FD will be such that either existing formal derivation (transformation, enrichment and refinement) techniques apply readily, or that it will spur the development of such techniques.

(9) Validation: a study will or might be conducted into the feasibility of the Ada FD serving as a basis for deriving "ACVC-like" test programs, and a study might be made into the feasibility of using the Ada FD to prove properties of ACVC-like programs. No attempt will, however, be made in this project to examine any serious fraction of the ACVC test suite for conformance to the Ada FD (or vice versa!). Thus, it is not within the scope of this project to study other than the "basic" aspect of section 5.9 and point (1) of the same sections' conformance part, and to study feasibility of its points (2-3-4).

(10) Proof Systems: a study might/will be made of the feasibility of deriving (a) proof system(s) from the Ada FD. It is, however, not a requirement that the currently planned Ada FD must be guaranteed to yield such (a) proof system(s).

(11) Mechanizable: the current Ada FD together with the LRM, its correlation to the LRM, its LRM-independent annotation, and various, not necessarily all-including, aspects of the underlying semantics of its specification language(s) will be mechanized. That is: a computerized tool set will be developed for the support of the activities mentioned in section 5.11.

(12) Prototyping: it is not a requirement of this project that the current Ada FD become the basis, or be shown feasible as a basis, for the rapid prototype development of, say, an interpreter.

But, along the lines of point (8) above, it is of interest to the current project to ascertain the extent of correlation between the Ada FD and the NYU/Ada ED interpreter written in SETL [30]. The current project does, however, not provide for a study of this, but would, in case such a study was undertaken, be most willing to co-operate, including striving to achieve "correlation".

(13) Correlation: it is a definite requirement that the developed Ada FD indeed be strongly, clearly, transparently and completely correlated to the LRM.

(14) Document Derivation: it is likewise (to point (13) above) a definite requirement, imposed by the developers, themselves, that the Ada FD be so expressed (presented) that it lends itself nicely to the derivation of a number of reference manuals for different levels (naive, novice, mature, experienced and sophisticated) of programmers, implementors, scientists (formalists), etc. The current project, however, only calls for one such informal document to be systematically derived.

(15) Maintability: it is a definite requirement that the current Ada FD be maintainable and as a derived requirement we find that point (11) then arises!

7. Conclusion

We have performed cursory and enumerative, rather than deep and analytical, studies of a number of classes of aspects of Ada, each leading up to our enumerations, qualifying and quantifying a number of requirements that we would wish the currently developing Ada FD to satisfy.

We submit this overview study to the international Ada community for its careful and co-operative scrutiny. We invite serious comments, and humbly expect both negative and positive critique. We declare ourselves most ready to seriously evaluate all comments for their proper disposal (including inclusion in a possibly reworked final version of this report) referred to, in the abstract, as (II).

8. REFERENCES

[1] Requirements to/of a Formal Definition of Ada
Dines Bjørner
Copenhagen, February, 1983. 4 pages

[2] What should be in a Formal Definition of Ada
R. Dewar, P. Kruchten, E. Schonberg
New York, 7 December, 1983, 6 pages

[3] Demonstrating Conformance between Formal Definition of Ada and the ACVC test suite
R. Dewar, P. Kruchten, E. Schonberg
New York, 6 December, 1983, 3 pages

[4] Requirements of a Definition of Ada
Andrew D. McGettrick
Glasgow, no date, 4 pages

[5] Role of the Formal Definition of Ada
Bernard Lang,
INRIA, no date, 10 pages

[6] The Users of a Formal Definition for Ada
Bernd Krieg-Brückner
2 April, 1983, 9 pages

[7] Working Paper for Formal Definition Working Group
Peter Wallis
Bath, no date, 1 page

[8] Joint Ada-Europe/ADA-TEC Meeting
Panel Discussion on the Formal Definition of Ada
18 March, 1982, 6 pages

[9] The DDC Formal Definition of Ada
Dines Bjørner
Copenhagen, no date, 7 pages

[10] Ada Formal Definition - Position Statement
Ole Oest
Copenhagen, 14 April, 1983, 3 pages

[11] Formal Definition of Ada, summary of Proposal
Dines Bjørner
Copenhagen, 10 May, 1983, 3 pages

[12] Proposal for a Co-operative European Effort for a Formal Definition of Ada
Bernd Krieg-Brückner, Georg Winterstein
Germany, May, 1983, 11 pages

[13] On the Timing of a Formal Definition for "Ada" (TM)
C. B. Jones
3 pages

[14] Personal Views on the Feasibility, Problems and Timing of the Ada Formal Definiton Effort
Joseph Stoy
Great Britain, 2 pages

[15] How to complete the Ada Formal Definition (draft)
Georg Winterstein
no date, 3 pages

[16] Note on A European VDM-based Formal Definition of Ada
Dines Bjørner
Copenhagen, 18 June, 1982, 15 pages

[17] An Abstract Systems Model of Ada Semantics
E. K. Blum
15 August, 1984

[18] D. Luckham et al.: ANNA: Annotated Ada

[19] J. Garwick
...

[20] Ada Compiler Validation Implementors' Guide
Softech Inc., 1980

[21] Towards a Formal Description of Ada
D. Bjørner, O. N. Oest
Springer Verlag
Lecture Notes in Computer Science, vol. 98, 1980

[22] Formal Definition of the Ada Programming Language
Preliminary Version for Public Review
Honeywell Inc., Cii Honeywell Bull and INIRA
November 1980

[23] Reference Manual for the Ada Programming Language
ANSI/MIL-STD 1815A
U.S. Department of Defense, Washington D.C.
January 1983

[24] Rationale for the Design of the Ada Programming Language
Draft for Editorial Review, Honeywell and Alsys
January 1984

[25] A Formal Definition of CHILL. A Supplement to the CCITT
Recommendation Z.200
Peter Haff, Dines Bjørner
Dansk Datamatik Center, 1980

[26] CHILL Language Definition
CCITT Recommendation Z.200, 1984

[27] Revised Report on the Algorithmic Language
ALGOL 60
P. Naur
Comm. ACM, vol 6, No. 1, pp 1 ff, 1963

[28] SIS: A Compiler Generator System using Denotational Semantic Definitions of Programming Language, Report ISI/RR-83-112,
Information Sciences/Institute, California

[30] An Executable Semantic Model for Ada, Ada/Ed Interpreter
Ada Project
Courant Institute, N.Y.U.

[31] Denotational Semantics
Joseph E. Stoy
MIT Press, 1977

[32] A Structured Approach to Operational Semantics
G. D. Plotkin
University of Aarhus, 1981

[33] On the Parameterized Algebraic Specification of Concurrent Systems
E. Astesiano, G. F. Mascari, G. Reggio and M. Wirsing
TAPSOFT Conf., Berlin, March 1985, pp 342-358
Springer LNCS vol 185

[34] Method and Notation for the Formal Definition of Programming Languages
Peter Lucas et al.
IBM Laboratory, Vienna
TR 25.087, Revised 1 July 1970

[35] Algebraic Semantics: Initial Semantics and Equations
H. Ehrig and B. Mahr.
Springer Verlag, EATCS Series, vol 5, 1985

[36] The Axiomatic Basis of Computer Programming,
C. A. R. Hoare
CACM, vol. 12, no. 10, pp 567-583, Oct. 1969

[37] An Axiomatic Definition of the Programming Language Pascal, C.A.R.Hoare and N. Wirth: Acta Informatica, vol. 2, pp 335-355, 1973

[39] Formalization in Program Development, P. Naur BIT, vol. 22, pp 437-453, 1982

[40] Mathematics, the Loss of Certainty, Morris Kline Oxford, University Press, 1982

[42] Calculus of Communication Systems, A.J. R. Milner Lecture Notes in Computer Science, vol. 94, Springer, 1980

[43] Communicating Sequential Processes, C. A. R. Hoare Prentice Hall Intl., 1985

[44] Processes and the Denotional Semantics of Concurrency Jaco de Bakker and Zuker Information and Control 54, 70-120 (1982)

[45] Natural Semantics on the Computer, G. Kahn et al. INRIA internal report, 24 May 1985.

[46] A Design for a SEMANOL Specification for Ada, TRW Report, 7 April 1980, and: A Multi-Processing Implementation-Oriented Formal Definition of Ada in Semanol, ACM SIGPLAN Symp. on the Ada Programming Language, Boston, Mass., SIGPLAN Notices, vol. 15, no. 11, Nov. 1980. F. C. Belz, E. K. Blum, and D. M. Heimbigner

[47] Formal Development of Interpreters and Compilers, ch.9 of Formal Specification and Software Development,
D. Bjørner
Prentice Hall International, 1982.

Appendix A: MNEMONICS

Once scientific ideas reach the market place their abstract nature gets instantiated, and commerical abbrevations result. Even in Academica we see such an unfortunate trend (e.g. ACT/ONE, ANNA, CCS, Cnet, CSP, DIANA, DSL, LARCH, LCS, Meta-IV, ML, OBJ, SIS, SMoLCS, and SOS). Bureaucracies foster "mnemoniconiae" (i.e. ANSI, BSI, CEC, CHILL, CNR, DoD, ECMA, IEI, ISO, LCB, LMC, LRM, MAP, MIL-STD, and WG). To add insult to injury we add our own: CRAI, DDC, FD, and MTL!

Abbrevations

ACT/ONE

ACVC Ada Compiler Validation Capability

ADT Abstract Data Type (usually algebraically specified).

ANNA ANNotated Ada
(D. Luckham et al.).

ANSI American National Standards Institute.

ASL Algebraic Specification Language
(D. Sanella and M. Wirsing).

BNF Backus Normal Form context free grammar.

BSI British Standards Institute.

CCITT Comité Consultatif International de Telegraphie et Telephonie.

CCS Calculus of Communication Systems
(R. Milner).

CEC The Commission of the European Communities.

CHILL CCITT High Level Language.

CLEAR (not a mnemonic, just a funny)
Name of a calculus for combining algebraically specified ADTs.
(R. Burstall and J. A. Goguen).

Cnet Campus net (Italian CNR project).

CNR Consiglio Nazionale della Ricerche
(Italian Council for National Research).

CRAI Consorzio per Ricerca e le Applicazioni de Informatica.

CSP Communicating Sequential Processes
(C.A.R. Hoare).

DDC Dansk Datamatik Center.

DIANA Descriptive Intermediate Attributed Notation for Ada

DoD (US) Department of Defense.

DSL Denotational Semantic Language
(P.D. Mosses).

D-SMoLCS Denotational SMoLCS.

ECMA European Computer Manufacturers Association.

FD Formal Definition.

IEI Istituto di Elaborazione della Informazione.

ISO International Standards Organisation.

LARCH (Not a mnemonic, but a) Name for an algebraic (semantics) specification language (family)
(J. Guttag and J. Horning).

LCB Language Control Board.

LCF Logic for Computable Functions (R. Milner).

LCS Labelled Control Systems.

LES Labelled Event Systems.

LMC Language Maintenance Committee.

LRM Language Reference Manual.

MAP Multi-Annual Programme in the field of data processing (of the CEC).

MENTOR

Meta-IV (Not really a mnemonic:) Meta-Language number four (rhymes with: Metaphor) (H. Bekič, D. Bjørner, C.B. Jones, P. Lucas).

MIL-STD MILitary STandarD.

NYU New York University.

ML Meta Language (as in Edinburgh/LCF) (R. Milner).

MTL Methods, Techniques, Languages (Ada FD)

OBJ Algebraic semantics OBJect specification language.

SETL SET (Programming) Language (NYU).

SIS Semantics Implementation System

(P.D. Mosses).

SMoLCS Structured, Monitored Linear Concurrent Systems
(E. Astesiano).

SOS Structural Operational Semantics
(G.D. Plotkin).

Appendix B: TERMINOLOGY

In any project, one should start out by carefully establishing and defining the terminology, and throughout the project one should critically maintain and adhere to this terminology. The terms name the important concepts. Hence, the terminology should also be part of the product.

The present section outlines only a very embryonic form of a terminology. It is part of a continuing activity of separately establishing a larger, more comprehensive, terminology document.

TERMS

Abstract Syntax

Definition of a class of objects which emphasizes their contents and structural relationship. In contrast to a concrete syntax an abstract syntax ignores the choice of lexical elements and their ordering in sentences of a language - more specifically a set of domain equations or a set of predicates, which define classes of abstract object.

Assertional (Pre-Post Specification) Language

An assertional specification language is one in which functions and operations may be defined by predicates over their arguments and results.

(In general, this technique defines a relation, and it is understood that such a relation is satisfied by any sub-relation with the same domain, in particular by a function with the same domain which is a sub-relation).

Combinator

A combinator is a syntactic rule which, when applied to a (usually fixed) number of formal documents, produces a resulting formal document. (Typically, a combinator is one or more symbols with rules for positioning input formal documents in relation to its symbol(s) to form a new document with a defined semantics.) Since the result of applying a combinator is a formal document, there has to be a rule which gives the semantics of the resulting document.

Concrete Syntax

A syntax which includes the definition of lexical elements and their ordering.

Constructive Specification

--same as model-oriented specification.

Correctness

The concept of correctness only has meaning in a context of a formal method which requires the generation of at least one pair of formal documents, A and B, so that A is considered to be a prescription for the production of B. B is then correct if it satisfies its prescription A. Typically, there will be several series of pairs (A,B), (B,C), (C,D), etc.

Data Type

A collection of objects, and operations involving these (and possibly other) objects.

Decomposition

A transformation from A to B so that there are functions/operations in A the behaviour of which is specified by a composition of functions/operations in B.

Deductive Specification

--same as property oriented specification.

Design

The supplier's statement of how the specification will be implemented. Such statements may exist at various levels of detail.

In the context of a formal method, a design is a transformation of a specification This transformation embodies decisions as to how the specification will be implemented.

Document

Any identifiable, finite piece of recorded information produced during a software development. A document may be expressed in a formal language or not, and may be electronically recorded or recorded by other means. (Thus, a program, a specification, and doodle are all documents if recorded and identifiable during a software development).

Enrichment

A transformation which includes the addition of functionality; in algebraic or denotational terms this could entail the addition of operations.

Fault Tolerance

Software is fault tolerant if it behaves correctly despite spurious failures in its input.

Formal Development

A development in which each generation of a transformation or implementation is accompanied by a formal proof of its correctness.

Formal (Document)

A document expressed in a formal language.

Formal (Language)

Having a precise semantics and syntax. (The syntax may be abstract).

Formal Method

A (software development) method whose guidelines are formalized and which requires the production of specifications in a formal language, in addition to the implementation (1).

Formal Proof

A proof in which each step is the application of an axiom of the inferential system or a formally proved theorem. The result of each step is expressed in a formal language.

Functional Specification

A functional specification is one which describes and prescribes the behaviour of its acceptable implementations in the following restricted sense:

(a) The only behaviour described is properties of the information content of the information input to and output from the implementation. No reference can be made to any other information, such as the passage of time, the internal representation in the implementation, etc.

(b) The only properties which are described are those which can be mathematically described or modelled,

(c) The claim that the implementation possesses these properties must be subject to refutation.

Generic Specification

A formal document which defines a class of specifications, (A parameterised specification is an example).

Genericity

Genericity is a general principle comprising an attribute of a document, process, method or other concept. It is the attribute of requiring a small fixed-size change to the document, process, etc., in order to achive a change in applicability of the doucment, process, etc.

Implementation

An executable specification which fulfils all the requirements.

Interpretation/Symbolic Execution

The interpretation or symbolic execution of a formal specification or an abstract design consists of a mechanically automated process of displaying properites of its implementations.

Loose Specification

--same as Generic Specification.

Maintainability

Software is maintainable if the insertion of a change can be unambiguously and uniquely located through specification design, and implementation.

Method

A method is a set of guidelines or rules for how to carry out a process (e.g. software development). Typically, the guidelines refer to specific tools which are to be applied using certain techniqued and in a prescribed order.

Rôle of the Ada Formal Definition: Terminology

Methodology

The science of methods. It is here used to denote a framework for a class or set of such methods.

Model Oriented Specifications

A specification (or design) which denoted a (theory of) mathematical model(s), i.e. an object, or a class of objects guaranteed to exist.

(In a model oriented specification language the specification language date types are typically definable in the following way. There are a fixed finite number of basic types supplied in the language. Their definitions may be axiomatic in style. Further data types may then be defined by applying type combinators of which there is a fixed repertoire in the language, in a possibly recursive manner.)

Modular(ised) Specification

A specificaition is modular(ised) if it is expressed as the composition of specifications.

Non-functional Specification

A specification which prescribes that its implementations shall possess a set of properites which do not comprise a functional specification (q.v.)

Parameterised Specification

Consider a specification combinator to the following form. The combinator is a formal document containing place-holders. The formal document becomes a specification if and only if the place-holders are associated with and semantically represent other specifications. Such a specification combinator is called a parameterised specification.

Performance

Performance is a quality of software which is not expressible within its functional specifications. The performance of software is the economy with which it exercises the resources of its environment. (Such resources are typically computer storage and c.p.u. time but, may be extended to include resources of a wider environment, such as fuel consumption of a software controlled industrial process.)

Programming

The activities involved in requirements definition, specification, design and implementation.

Property Oriented Specifiaction

A specification is property oriented if it defines the external characterisitcs only.

A property oriented specification language is one which permits the definition of data types by defining new functions in the sorts of which the data type to be defined occurs, and then listing properties of the new functions. This in general defines a class of data types, and there will be defined an "interpretation" which will identify (to within isomorphism) a unique data type from this class. An alternative to the last provision is that of a loose interparation in which the data type is not further identitied. The specification which reusults is then parameterised, and will become a proper specification when an instatiation of the data type is given which define it uniquely (to within isormorphism).

Properiety, Proper

Propriety (adjective: proper) is the attribute of software of fulfilling the functional and non-functional (q.v. expectations of its users. (If the requirements documents have been adequately formulated, they should be caputured therein.)

Prototype

An executable model that conforms to a subset, or is a projection of the requirements of the specification.

Prototyping

The act of constructing a prototype. Typically involved transliterating a specification or an abstract design into some high level executable code.

Refinement

A structure preserving decomposition of specification A to (a possibly abstract) design B, or, similarly, from design B to design C, etc.

Reliability

Software is reliable if it is both correct and also clearly rejects input explicity excluded from is specification.

Requirements

The customer's statement of his needs.

Requirements Analysis

An analysis of the customer's needs, cf. contractual model.

Rigorous Development

A development in which each generation of a transformation, refinement, or enrichment gives rise to a proof obligation, which can be accompanied by a rigorous proof.

Rigorous Proof

A rigorous proof is a demonstration designed to convince others (that a formal proof could be generated) of the truth of some assertion.

Robustness

Software is robust if changes to it do not hamper its quality, i.e. its conformance to its functional and non-functional specifications (q.v.).

Satisfy

A formal specification B satisfies a formal specification A if B exhibits all the behaviour specified by A.

Software Engineering

The total support process of producing and delivering implementations and maintaining them, starting from requirements.

Specification

The supplier's description of the functional behaviour of the implementation and the process of producing it.

Specification Composition

The process or result of transitively applying a number of combinators to a number of specifications to produce a resulting specification.

Specification Language

A language in which specifications can be expressed.

Specialization of Specifications

The process of transformning a generic specification into a formal document whih defines a sub-class (possibly one) of specifications.

Support System

An integrated collection of tools supporting some particular kind of activity in the software field. The activity may be broad or narrow.

Symbolic Execution

--same as interpretation.

Syntax

A definition (usually formal) of the allowable sentences of a language.

Systematic

A systematic method (as oppoesd to a rigorous or formal one) is one comprisinfg rules and/or guidelines for the ordered production of (mostly informal) documents within a development process. A systematic development is one carried out according to the rules and/or guidelines of a systematic method.

Testing (1)

The systematic and organized search for a counter-example to the claim that a specification/design/implementation is correct.

Testing (2)

The (possibly partial) execution of a specification in order to demonstrate that it fulfils some non-functional requirements (and hence to demonstrate its propriety).

Tool

An object that can be used in the process of developing (and maintaining) software systems. Tools include both computerized as well as non-computerized (manual) tools.

Transformation

The process or result of generation a formal specification B from a given formal specification A such that B satisfies A, where the internal structure of B is normally not a decomposition of that of A. The term is often used in the context of a machine-generated or machine-assisted transformation.

Wherever no specific qualification is made, transformation will include the concepts of refinement and enrichment.

User

Someone who uses Ada FD.

Validation

A process within the contractual view of software development which improves confidence in the correctness of a specification, design, or implementation, or the claim that a specification, etc. fulfils the requirements. This could be the production of a proof in the former case.

Verification

A proof that a transformation, enrichment, or refinement is correct.

Or, alternatively: A process within the contractual view of software development which improves confidence in the well-formedness and non-degeneracy of a specification/design/implementation. This could be the production of a proof.

DISTRIBUTION LIST OF M-135

Bernard Abrams
Grumman Aerospace Corporation
Mail Station 001-31T
Bethpage, NY 11714
(516) 575-9487
ABRAMS@USC-ECLB

* Omar Ahmed
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

* Eric R. Anderson
TRW DSG (R2/1134)
One Space Park
Redondo Beach, CA 90278
(213) 535-5776
TRWRB!TRWSPP!ERA@BERKELEY

* Dr. Thomas C. Antognini
MITRE Corporation
Mailstop B330
Burlington Road
Bedford, MA 01730
(617) 271-7294
SECURITY!TCA@MITRE-BEDFORD or
TCVB@MITRE-BEDFORD

Charles Applebaum
1058 Boyurgogne
Bowling Green, OH 43402
(419) 352-0777
CHA@MITRE-BEDFORD

Krzystof Apt
Thomas J. Watson Research Center
P. O. Box 218
88-K01 Route 134
Yorktown Heights, NY 10598
(914) 945-2923

Terry Arnold
Merdan Group
P.O. Box 17098
San Diego, CA 92117
MERDAN@ISI

Ted Baker
Department of Computer Science
Florida State University
Tallahassee, FL 32306
(904) 644-2296

David Elliot Bell — DBELL@MIT-MULTICS
Trusted Information Systems, Inc.
3060 Washington Road
Glenwood, MD 21738
(301) 854-5889

Dan Berry
3531G Boelter Hall
Computer Science Department
School of Eng. and Appl. Science
Los Angeles, CA 90024
(213) 825-2971

Edward K. Blum — BLUM@ECLB
Mathematics Department
University of Southern California
Los Angelos, CA 90089
(213) 743-2504

* Alton L. Brintzenhoff — SCI-ADA@USC-ISI
SYSCON Corporation
3990 Sherman Street
San Diego, CA 92110
(619) 296-0085

* Dr. Dianne Britton — HELBIG@ISI
RCA Adv. Tech. Labs
ATL Building
Moorestown Corporate Center
Moorestown, NJ 08057
(609) 866-6654 or (609) 924-3253

* Dr. R. Leonard Brown — BROWN@AEROSPACE
M1/112
The Aerospace Corporation
P. O. Box 92957
Los Angeles, CA 90009
(213) 615-4335

Richard Chan — RCHAN@USC-ECL (bad)
Hughes Aircraft Co.
P. O. Box 33
FU-618/P115
Fullerton, CA 92634
(714) 732-7659

* Norman Cohen — NCOHEN@ECLB
SofTech, Inc.
705 Masons Mill Business Park
1800 Byberry Road
Huntingdon Valley, PA 19006
(215) 947-8880

* Paul M. Cohen — PCOHEN@ECLB
Ada Joint Program Office
OUSDRE/R&AT
Pentagon Room 3D139 (Fern Street)
Washington, DC 20301-3081
(202) 694-0211

Richard M. Cohen — COHEN@UTEXAS-20
Institute for Computing Science
2100 Main Bldg.
University of Texas
Austin, Texas 78712
(512) 471-1901

Michael D. Colgate — FREEMAN@FORD-COS1
Ford Aerospace & Comm. Corp.
10440 State Highway 83
Colorado Springs, Colorado 80908

* Mark R. Cornwell — CORNWELL@NRL-CSS
Code 7590
Naval Research Lab
Washington, D.C. 20375
(202) 767-3365

Major Terry Courtwright — COURT@MITRE
WIS/JPMO/ADT
7726 Old Springhouse Road
Washington, DC 20330-6600
(202) 285-5056

* Dan Craigen — CMP.CRAIGEN@UTEXAS-20
c/o I. P. Sharp Associates
265 Carling Avenue
Suite 600
Ottawa, Ontario, Canada K1S 2E1
(613) 236-9942

Steve Crocker, M1-101 — CROCKER@AEROSPACE
The Aerospace Corporation
P.O. Box 92957
Los Angeles, CA 92957
(213) 648-6991

John J. Daly — WCOXTON@USADHQ2
USAISSAA
2461 Eisenhower Avenue
Alexandria, VA 22331-0700

Tom Dee
Boeing Commercial Airplane Co.
P. O. Box 3707
MS 77-21
Seattle, WA 98124
(206) 237-0194

Jeff Facemire
Texas Instruments
P.O. Box 801
M/S 8007
2501 West University
McKinney, TX 75069
(214) 952-2137
FACEMIRE%TI-EG@CSNET-RELAY

* John C. Faust
RADC/COTC
Griffiss AFB, NY 13441
(315) 330-3241
FAUST@RADC-MULTICS

Gerry Fisher
IBM Research 35-162
P. O. Box 218
Yorktown Heights, NY 10598
(914) 945-1677

Roy S. Freedman
Hazeltine Corporation
Greenlawn, NY 11740
(516) 261-7000
FREEDMAN@ECLB

James W. Freeman
Ford Aerospace & Comm. Corp.
Mailstop 15A
10440 State Highway 83
Colorado Springs, CO 80908
(303) 594-1536

Mark Gerhardt
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7839
MSG@MITRE-BEDFORD

Chuck Gerson
Boeing Aerospace Co.
Mailstop 8H-56
P.O. Box 3999
Seattle, WA 98124

Helen Gill
MITRE
Mailstop W459
1820 Dolly Madison Boulevard
McLean, Virginia 22102
(703) 883-7980

Kathleen A. Gilroy
Software Prod. Solutions, Inc.
P. O. Box 361697
Melbourne, FL 32936

Virgil Gligor
Department of Electrical Engineering
University of Maryland
College Park, Maryland 20742
(301) 454-8846

Donald I. Good GOOD@UTEXAS-20
2100 Main Building
The University of Texas at Austin
Austin, TX 78712
(512) 471-1901

Ronald A. Gove GOVE@MIT-MULTICS
Booz, Allen & Hamilton
4330 East West Highway
Bethesda, MD 20814
(301) 951-4624

* Inara Gravitis GRAVITIS@ECLB
SAIC
1710 Goodridge Drive
McLean, VA 22202
(703) 734-4096 or (202) 697-3749

* Col. Joseph S. Greene, Jr. JGREENE@USC-ISI
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6818

David Gries GRIES@CORNELL
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
(607) 256-4052

David Guaspari
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020
RPLATEK@ECLB

* J. Daniel Halpern
SYTEK Corp.
1225 Charleston Road
Mountain View, CA 94043
(415) 966-7300
SYTEK@SRI-UNIX or
MENLO70!SYTEK!DAN@BERKELEY

* Kurt W. Hansen
Dansk Datamatik Center
LuudToftevej 1C
DK2800 Lyngby
Denmark
PHONE: ++ 45 2 872622
KHANSEN@ECLB

* Scott Hansohn
Honeywell Secure Comp. Tech. Center
Suite 130
2855 Anthony Lane South
St. Anthony, MN 55418
(612) 379-6434
HANSOHN@HI-MULTICS

* Larry Hatch
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6790
HATCH@TYCHO

Linn Hatch
IBM
17100 Frederick Heights
Gaithersburg, MD 20879

* Brian E. Holland
DoDCSC, C3
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6968
BRIAN@TYCHO

Ray Hookway
Dept. of Computer Eng. & Science
Case Institute of Technology
Case Western Reserve University
Cleveland, OH 44106
(216) 368-2800
HOOKWAY%CASE@CSNET-RELAY

Paul Hubbard
Dept. of Computer Eng. & Science
Case Institute of Technology
Case Western Reserve University
Cleveland, OH 44106
(216) 368-2800
HOOKWAY%CASE@CSNET-RELAY

Jim Huitema
National Security Agency
R831
Ft. Meade, MD 20755
(301) 859-6921

Larry A. Johnson
GTE
77 "A" Street
Needham, MA 02194
(617) 449-2000 ext. 3248
LJOHNSON@MIT-MULTICS

* Juern Juergens
SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02254
(617) 890-6900 ext. 316
JJURGENS@ECLB

Matt Kaufmann
Burroughs Corp.
Austin Research Center
12201 Technology Blvd.
Austin, TX 78727
(512) 258-2495
CMP.BARC@UTEXAS-20

Prof. Richard A. Kemmerer
Computer Science Department
University of California
Santa Barbara, CA 93106
(805) 961-4232
DICK@UCLA-CS

John C. Knight
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22903
(804) 924-1030
UVACS!JCK@SEISMO

Major Al Kopp
Ada Joint Program Office
OUSDRE/R&AT
Pentagon Room 3D139 (Fern Street)
Washington, DC 20301-3081
(202) 694-0211
AKOPP@ECLB

* Thomas M. Kraly
IBM Federal Systems Division
Software Eng. & Tech. 4D08
6600 Rockledge Drive
Bethesda, MD 20817
(301) 493-1449

Dr. Jack Kramer KRAMER@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2263

Eduardo Krell
3804 Boelter Hall
UCLA
Los Angeles, CA 90024

Kathy Kucheravy
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755

Dr. Kenneth Kung KKUNG@USC-ECLA
Hughes Aircraft Company
Ground Systems Group
M. S. 618/Q315
P. O. Box 3310
Fullerton, CA 92634
(714) 732-0262

* Carl Landwehr LANDWEHR@NRL-CSS
Code 7593
Naval Research Laboratory
Washington, DC 20375-5000
(202) 767-3381

* Mike Lake MLAKE@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Division
Alexandria, VA 22311
(703) 845-2519

Randall E. Leonard
Army Sys. Software Support Command
ATTN: ASB-QAA
Fort Belvoir, VA 22060

Nancy Leveson
ICS Department
University of California
Irvine, CA 92717
(714) 548-7525 or (714) 856-5517

Dr. Timothy E. Lindquist — LINDQUIS%ASU.CSNET@CSNET-RELAY
Computer Science Department
Arizona State University
Tempe, AZ 85287
(602) 965-2783

* Steven Litvintchouk — SDL@MITRE-BEDFORD
Mail Stop A180T
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7753

* David Luckham — LUCKHAM@SAIL
Stanford University
Computer Systems Lab, ERL 456
Stanford, CA 94305
(415) 497-1242

Dr. Glenn MacEwen
Computing and Information Science
Goodwin Hall
Queens University
Kingston, Ontario
K7L 3N6
(613) 547-2915 or (613) 548-4355

* Ann Marmor-Squires — MARMOR@ISI
TRW
Defense Systems Group
2751 Prosperity Avenue
Fairfax, VA 22031
(703) 876-8170

Eric Marshall — PAYTON@BBNG
System Development Corporation
P.O. Box 517
Paoli, PA 19301
(215) 648-7223

* Adrian R. D. Mathias — RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

* Terry Mayfield — TMAYFIELD@ECLB
Institute for Defense Analyses
Computer & Software Division
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2479

* John McHugh — MCHUGH@UTEXAS-20
Research Triangle Institute
Box 12194
Research Triangle Park, NC 27709
(919) 541-7327

Rudolf W. Meijer — RMEIJER@USC-ECLB
Commission of the European Communities
Info. Tech. and Telecomm. Task Force
A25 9/6A
Rue de la Loi 200
B-1049 Brussels, Belgium
PHONE: +32 2 235 7769

* Donn Milton — VRDXHQ!DRM1@SEISMO
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

* Warren Monroe — WMONROE@ECLA
Hughes Aircraft Co.
P.O. Box 3310
FU-618/Q315
Fullerton, CA 92634
(714) 732-2887

Mark Moriconi — MORICONI@SRI-CSL
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-5364

* LCDR Philip A. Myers — MYERS@NRL-CSR
Space and Naval Warfare Sys. Command
SPAWAR 8141A
Washington, DC 20363-5001
(202) 692-8484

* Karl Nyberg — NYBERG@ECLB
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

* Myron Obaranec — LAKSHMI@CECOM-1
U. S. Army, CECOM
Fort Monmouth, NJ 07703
ATTN: AMSEL-TCS-SIO
(201) 544-4962

Frank J. Oles
Thomas J. Watson Research Center
P.O. Box 218
88-K01 Route 134
Yorktown Heights, NY 10598
(914) 945-2012

Mahmoud Parsian
SDI Inc.
P. O. Box 4283
Falls Church, VA 22044

Diana B. Parton — DBP@MITRE-BEDFORD
The MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7754

* Don Peters
Comm. Sec. Establishment
Dept. of Nat. Defence
101 Colonel By Drive
Ottawa K1A 0K2 CANADA
(613) 998-4519

* John Peterson — PETERSON@TYCHO
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755
(301) 859-6790

* Joseph E. Pfauntsch, MS 29A — JEP@FORD-COS4
Ford Aerospace & Comm. Corp.
10440 State Highway 83
Colorado Springs, Colorado 80908
(303) 594-1326

* Richard Platek — RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

Erhard Ploedereder
Tartan Labs
411 Melwood Avenue
Pittsburgh, PA 15213
(412) 621-2210
PLOEDEREDER@TARTAN

* David Preston
IITRI
5100 Forbes Blvd.
Lanham, MD 20706
(301) 459-3711
DPRESTON@ECLB

Sri Rajeev
AT&T Bell Laboratories
Room 1-342
190 River Road
Summit, NJ 07901
(201) 522-6330
IHNP4!ATTUNIX!RAJEEV@BERKELEY

* William D. Ricker
The MITRE Corporation
M/S K229
Burlington Road
Bedford, MA 01730
(617) 271-3001
WDR@MITRE-BEDFORD

* R. Max Robinson
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2097
RROBINSON@USC-ECLB

W. A. Robison
30 Charles Street West
Apt. # 1811
Toronto, Ontario, CANADA
M4Y 1R5
(416) 925-0751

* Clyde G. Roby
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2541
CROBY@ECLB

Ken Rowe
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755

John Rushby - EL393
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-5456

RUSHBY@SRI-CSL

* Mark Saaltink
I. P. Sharp Associates
265 Carling Avenue
Suite 600
Ottawa, Ontario, Canada K1S 2E1
(613) 236-9942

SAALTINK@MIT-MULTICS

Marvin Schaefer
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6880 or (301) 859-6818

SCHAEFER@USC-ISI

* Mike Schwartz
Mailstop L0402
Martin-Marietta
Denver Aerospace
P. O. Box 179
Denver, CO 80201
(303) 977-0421

UCBVAX!HPLABS!HAO!DENELCOR!

Dev Sen
STC IDEC LIMITED
Technology Division
Six Hills House
London Road
Stevenage
Hertfordshire S61 1YB ENGLAND
PHONE: 011-44-438-726161

* Jerry Shelton
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

VRDXHQ!JHS@SEISMO

* Brian Siritzky (212) 460-7239
Dept. of Computer Science
Courant Institue of Math. Sciences
New York University
251 Mercer Street
New York, NY 10012

SIRITZKY@NYU-ACF2 or
...CMCL2!ACF2!SIRITZKY

Doug Weber
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020
RPLATEK@ECLB

* Steve Welke
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2393
SWELKE@ECLB

Col. William Whitaker
WIS/JPMO/ADT
7726 Old Springhouse Road
Washington, DC 20330-6600
(202) 285-5065
WWHITAKER@ECLB

* Jim Williams
MITRE Corporation
Mailstop B332
Burlington Road
Bedford, MA 01730
(617) 271-2647
JGW@MITRE-BEDFORD

Jim Wolfe
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2109
JWOLFE@ECLB

Larry Yelowitz
Ford Aerospace and Comm. Corp.
Western Development Lab. Div.
Mailstop X-20
3939 Fabian Way
Palo Alto, CA 94303
(415) 852-4198
KLY@FORD-WDL1

* Christine Youngblut
Advanced Software Methods, Inc.
17021 Sioux Lane
Gaithersburg, MD 20878
(301) 948-1989
CYOUNGBLUT@ECLB

* Margie Zuk
Mailstop B321, Bldg B
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7590
MMZ@MITRE-BEDFORD

DISTRIBUTION LIST OF M-135

Bernard Abrams
Grumman Aerospace Corporation
Mail Station 001-31T
Bethpage, NY 11714
(516) 575-9487
ABRAMS@USC-ECLB

* Omar Ahmed
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

* Eric R. Anderson
TRW DSG (R2/1134)
One Space Park
Redondo Beach, CA 90278
(213) 535-5776
TRWRB!TRWSPP!ERA@BERKELEY

* Dr. Thomas C. Antognini
MITRE Corporation
Mailstop B330
Burlington Road
Bedford, MA 01730
(617) 271-7294
SECURITY!TCA@MITRE-BEDFORD or
TCVB@MITRE-BEDFORD

Charles Applebaum
1058 Boyurgogne
Bowling Green, OH 43402
(419) 352-0777
CHA@MITRE-BEDFORD

Krzystof Apt
Thomas J. Watson Research Center
P. O. Box 218
88-K01 Route 134
Yorktown Heights, NY 10598
(914) 945-2923

Terry Arnold
Merdan Group
P.O. Box 17098
San Diego, CA 92117
MERDAN@ISI

Ted Baker
Department of Computer Science
Florida State University
Tallahassee, FL 32306
(904) 644-2296

David Elliot Bell
Trusted Information Systems, Inc.
3060 Washington Road
Glenwood, MD 21738
(301) 854-5889

DBELL@MIT-MULTICS

Dan Berry
3531G Boelter Hall
Computer Science Department
School of Eng. and Appl. Science
Los Angeles, CA 90024
(213) 825-2971

Edward K. Blum
Mathematics Department
University of Southern California
Los Angelos, CA 90089
(213) 743-2504

BLUM@ECLB

* Alton L. Brintzenhof
SYSCON Corporation
3990 Sherman Street
San Diego, CA 92110
(619) 296-0085

SCI-ADA@USC-ISI

* Dr. Dianne Britton
RCA Adv. Tech. Labs
ATL Building
Moorestown Corporate Center
Moorestown, NJ 08057
(609) 866-6654 or (609) 924-3253

HELBIG@ISI

* Dr. R. Leonard Brown
M1/112
The Aerospace Corporation
P. O. Box 92957
Los Angeles, CA 90009
(213) 615-4335

BROWN@AEROSPACE

Richard Chan
Hughes Aircraft Co.
P. O. Box 33
FU-618/P115
Fullerton, CA 92634
(714) 732-7659

RCHAN@USC-ECL (bad)

* Norman Cohen
SofTech, Inc.
705 Masons Mill Business Park
1800 Byberry Road
Huntingdon Valley, PA 19006
(215) 947-8880

NCOHEN@ECLB

* Paul M. Cohen — PCOHEN@ECLB
Ada Joint Program Office
OUSDRE/R&AT
Pentagon Room 3D139 (Fern Street)
Washington, DC 20301-3081
(202) 694-0211

Richard M. Cohen — COHEN@UTEXAS-20
Institute for Computing Science
2100 Main Bldg.
University of Texas
Austin, Texas 78712
(512) 471-1901

Michael D. Colgate — FREEMAN@FORD-COS1
Ford Aerospace & Comm. Corp.
10440 State Highway 83
Colorado Springs, Colorado 80908

* Mark R. Cornwell — CORNWELL@NRL-CSS
Code 7590
Naval Research Lab
Washington, D.C. 20375
(202) 767-3365

Major Terry Courtwright — COURT@MITRE
WIS/JPMO/ADT
7726 Old Springhouse Road
Washington, DC 20330-6600
(202) 285-5056

* Dan Craigen — CMP.CRAIGEN@UTEXAS-20
c/o I. P. Sharp Associates
265 Carling Avenue
Suite 600
Ottawa, Ontario, Canada K1S 2E1
(613) 236-9942

Steve Crocker, M1-101 — CROCKER@AEROSPACE
The Aerospace Corporation
P.O. Box 92957
Los Angeles, CA 92957
(213) 648-6991

John J. Daly — WCOXTON@USADHQ2
USAISSAA
2461 Eisenhower Avenue
Alexandria, VA 22331-0700

Tom Dee
Boeing Commercial Airplane Co.
P. O. Box 3707
MS 77-21
Seattle, WA 98124
(206) 237-0194

Jeff Facemire — FACEMIRE%TI-EG@CSNET-RELAY
Texas Instruments
P.O. Box 801
M/S 8007
2501 West University
McKinney, TX 75069
(214) 952-2137

* John C. Faust — FAUST@RADC-MULTICS
RADC/COTC
Griffiss AFB, NY 13441
(315) 330-3241

Gerry Fisher
IBM Research 35-162
P. O. Box 218
Yorktown Heights, NY 10598
(914) 945-1677

Roy S. Freedman — FREEDMAN@ECLB
Hazeltine Corporation
Greenlawn, NY 11740
(516) 261-7000

James W. Freeman
Ford Aerospace & Comm. Corp.
Mailstop 15A
10440 State Highway 83
Colorado Springs, CO 80908
(303) 594-1536

Mark Gerhardt — MSG@MITRE-BEDFORD
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7839

Chuck Gerson
Boeing Aerospace Co.
Mailstop 8H-56
P.O. Box 3999
Seattle, WA 98124

Helen Gill
MITRE
Mailstop W459
1820 Dolly Madison Boulevard
McLean, Virginia 22102
(703) 883-7980

Kathleen A. Gilroy
Software Prod. Solutions, Inc.
P. O. Box 361697
Melbourne, FL 32936

Virgil Gligor
Department of Electrical Engineering
University of Maryland
College Park, Maryland 20742
(301) 454-8846

Donald I. Good — GOOD@UTEXAS-20
2100 Main Building
The University of Texas at Austin
Austin, TX 78712
(512) 471-1901

Ronald A. Gove — GOVE@MIT-MULTICS
Booz, Allen & Hamilton
4330 East West Highway
Bethesda, MD 20814
(301) 951-4624

* Inara Gravitis — GRAVITIS@ECLB
SAIC
1710 Goodridge Drive
McLean, VA 22202
(703) 734-4096 or (202) 697-3749

* Col. Joseph S. Greene, Jr. — JGREENE@USC-ISI
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6818

David Gries — GRIES@CORNELL
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
(607) 256-4052

David Guaspari
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020
RPLATEK@ECLB

* J. Daniel Halpern
SYTEK Corp.
1225 Charleston Road
Mountain View, CA 94043
(415) 966-7300
SYTEK@SRI-UNIX or
MENLO70!SYTEK!DAN@BERKELEY

* Kurt W. Hansen
Dansk Datamatik Center
LuudToftevej 1C
DK2800 Lyngby
Denmark
PHONE: ++ 45 2 872622
KHANSEN@ECLB

* Scott Hansohn
Honeywell Secure Comp. Tech. Center
Suite 130
2855 Anthony Lane South
St. Anthony, MN 55418
(612) 379-6434
HANSOHN@HI-MULTICS

* Larry Hatch
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6790
HATCH@TYCHO

Linn Hatch
IBM
17100 Frederick Heights
Gaithersburg, MD 20879

* Brian E. Holland
DoDCSC, C3
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6968
BRIAN@TYCHO

Ray Hookway
Dept. of Computer Eng. & Science
Case Institute of Technology
Case Western Reserve University
Cleveland, OH 44106
(216) 368-2800
HOOKWAY%CASE@CSNET-RELAY

Paul Hubbard
Dept. of Computer Eng. & Science
Case Institute of Technology
Case Western Reserve University
Cleveland, OH 44106
(216) 368-2800
HOOKWAY%CASE@CSNET-RELAY

Jim Huitema
National Security Agency
R831
Ft. Meade, MD 20755
(301) 859-6921

Larry A. Johnson
GTE
77 "A" Street
Needham, MA 02194
(617) 449-2000 ext. 3248
LJOHNSON@MIT-MULTICS

* Juern Juergens
SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02254
(617) 890-6900 ext. 316
JJURGENS@ECLB

Matt Kaufmann
Burroughs Corp.
Austin Research Center
12201 Technology Blvd.
Austin, TX 78727
(512) 258-2495
CMP.BARC@UTEXAS-20

Prof. Richard A. Kemmerer
Computer Science Department
University of California
Santa Barbara, CA 93106
(805) 961-4232
DICK@UCLA-CS

John C. Knight
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22903
(804) 924-1030
UVACS!JCK@SEISMO

Major Al Kopp
Ada Joint Program Office
OUSDRE/R&AT
Pentagon Room 3D139 (Fern Street)
Washington, DC 20301-3081
(202) 694-0211
AKOPP@ECLB

* Thomas M. Kraly
IBM Federal Systems Division
Software Eng. & Tech. 4D08
6600 Rockledge Drive
Bethesda, MD 20817
(301) 493-1449

Dr. Jack Kramer KRAMER@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2263

Eduardo Krell
3804 Boelter Hall
UCLA
Los Angeles, CA 90024

Kathy Kucheravy
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755

Dr. Kenneth Kung KKUNG@USC-ECLA
Hughes Aircraft Company
Ground Systems Group
M. S. 618/Q315
P. O. Box 3310
Fullerton, CA 92634
(714) 732-0262

* Carl Landwehr LANDWEHR@NRL-CSS
Code 7593
Naval Research Laboratory
Washington, DC 20375-5000
(202) 767-3381

* Mike Lake MLAKE@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Division
Alexandria, VA 22311
(703) 845-2519

Randall E. Leonard
Army Sys. Software Support Command
ATTN: ASB-QAA
Fort Belvoir, VA 22060

Nancy Leveson
ICS Department
University of California
Irvine, CA 92717
(714) 548-7525 or (714) 856-5517

Dr. Timothy E. Lindquist — LINDQUIS%ASU.CSNET@CSNET-RELAY
Computer Science Department
Arizona State University
Tempe, AZ 85287
(602) 965-2783

* Steven Litvintchouk — SDL@MITRE-BEDFORD
Mail Stop A180T
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7753

* David Luckham — LUCKHAM@SAIL
Stanford University
Computer Systems Lab, ERL 456
Stanford, CA 94305
(415) 497-1242

Dr. Glenn MacEwen
Computing and Information Science
Goodwin Hall
Queens University
Kingston, Ontario
K7L 3N6
(613) 547-2915 or (613) 548-4355

* Ann Marmor-Squires — MARMOR@ISI
TRW
Defense Systems Group
2751 Prosperity Avenue
Fairfax, VA 22031
(703) 876-8170

Eric Marshall — PAYTON@BBNG
System Development Corporation
P.O. Box 517
Paoli, PA 19301
(215) 648-7223

* Adrian R. D. Mathias — RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

* Terry Mayfield — TMAYFIELD@ECLB
Institute for Defense Analyses
Computer & Software Division
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2479

* John McHugh — MCHUGH@UTEXAS-20
Research Triangle Institute
Box 12194
Research Triangle Park, NC 27709
(919) 541-7327

Rudolf W. Meijer — RMEIJER@USC-ECLB
Commission of the European Communities
Info. Tech. and Telecomm. Task Force
A25 9/6A
Rue de la Loi 200
B-1049 Brussels, Belgium
PHONE: +32 2 235 7769

* Donn Milton — VRDXHQ!DRM1@SEISMO
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

* Warren Monroe — WMONROE@ECLA
Hughes Aircraft Co.
P.O. Box 3310
FU-618/Q315
Fullerton, CA 92634
(714) 732-2887

Mark Moriconi — MORICONI@SRI-CSL
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-5364

* LCDR Philip A. Myers — MYERS@NRL-CSR
Space and Naval Warfare Sys. Command
SPAWAR 8141A
Washington, DC 20363-5001
(202) 692-8484

* Karl Nyberg — NYBERG@ECLB
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

* Myron Obaranec
U. S. Army, CECOM
Fort Monmouth, NJ 07703
ATTN: AMSEL-TCS-SIO
(201) 544-4962
LAKSHMI@CECOM-1

Frank J. Oles
Thomas J. Watson Research Center
P.O. Box 218
88-K01 Route 134
Yorktown Heights, NY 10598
(914) 945-2012

Mahmoud Parsian
SDI Inc.
P. O. Box 4283
Falls Church, VA 22044

Diana B. Parton
The MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7754
DBP@MITRE-BEDFORD

* Don Peters
Comm. Sec. Establishment
Dept. of Nat. Defence
101 Colonel By Drive
Ottawa K1A 0K2 CANADA
(613) 998-4519

* John Peterson
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755
(301) 859-6790
PETERSON@TYCHO

* Joseph E. Pfauntsch, MS 29A
Ford Aerospace & Comm. Corp.
10440 State Highway 83
Colorado Springs, Colorado 80908
(303) 594-1326
JEP@FORD-COS4

* Richard Platek
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020
RPLATEK@ECLB

Erhard Ploedereder
Tartan Labs
411 Melwood Avenue
Pittsburgh, PA 15213
(412) 621-2210
PLOEDEREDER@TARTAN

* David Preston
IITRI
5100 Forbes Blvd.
Lanham, MD 20706
(301) 459-3711
DPRESTON@ECLB

Sri Rajeev
AT&T Bell Laboratories
Room 1-342
190 River Road
Summit, NJ 07901
(201) 522-6330
IHNP4!ATTUNIX!RAJEEV@BERKELEY

* William D. Ricker
The MITRE Corporation
M/S K229
Burlington Road
Bedford, MA 01730
(617) 271-3001
WDR@MITRE-BEDFORD

* R. Max Robinson
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2097
RROBINSON@USC-ECLB

W. A. Robison
30 Charles Street West
Apt. # 1811
Toronto, Ontario, CANADA
M4Y 1R5
(416) 925-0751

* Clyde G. Roby
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2541
CROBY@ECLB

Ken Rowe
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755

John Rushby - EL393
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-5456

RUSHBY@SRI-CSL

* Mark Saaltink
I. P. Sharp Associates
265 Carling Avenue
Suite 600
Ottawa, Ontario, Canada K1S 2E1
(613) 236-9942

SAALTINK@MIT-MULTICS

Marvin Schaefer
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6880 or (301) 859-6818

SCHAEFER@USC-ISI

* Mike Schwartz
Mailstop L0402
Martin-Marietta
Denver Aerospace
P. O. Box 179
Denver, CO 80201
(303) 977-0421

UCBVAX!HPLABS!HAO!DENELCOR!

Dev Sen
STC IDEC LIMITED
Technology Division
Six Hills House
London Road
Stevenage
Hertfordshire S61 1YB ENGLAND
PHONE: 011-44-438-726161

* Jerry Shelton
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

VRDXHQ!JHS@SEISMO

* Brian Siritzky (212) 460-7239
Dept. of Computer Science
Courant Institue of Math. Sciences
New York University
251 Mercer Street
New York, NY 10012

SIRITZKY@NYU-ACF2 or
...CMCL2!ACF2!SIRITZKY

* Roger Smeaton — SMEATON@NOSC-TECR
NOSC, Code 423
San Diego, CA 92152
(619) 225-2083

Michael Smith — MKSMITH@UTEXAS
ICSCA
2100 Main Building
University of Texas
Austin, TX 78712
(512) 471-1901

* Ryan Stansifer — RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

* David Sutherland — RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

Steve Sutkowski — INCO@USC-ISID
Inco Inc.
8260 Greensboro Drive
McLean, VA 22102
(703) 883-4933

Michael Thompson
Astronautics Corporation of America
P. O. Box 523
Milwaukee, Wisconsin 53201-0523
(414) 447-8200

* Friedrich von Henke — VONHENKE@SRI-CSL
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-2560

Barry Watson — WATSON@ECLB
Ada Information Clearinghouse
IITRI
Room 3D139 (1211 Fern St., C-107)
The Pentagon
Washington, DC 20301
(703) 685-1477

Doug Weber
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020
RPLATEK@ECLB

* Steve Welke
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2393
SWELKE@ECLB

Col. William Whitaker
WIS/JPMO/ADT
7726 Old Springhouse Road
Washington, DC 20330-6600
(202) 285-5065
WWHITAKER@ECLB

* Jim Williams
MITRE Corporation
Mailstop B332
Burlington Road
Bedford, MA 01730
(617) 271-2647
JGW@MITRE-BEDFORD

Jim Wolfe
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2109
JWOLFE@ECLB

Larry Yelowitz
Ford Aerospace and Comm. Corp.
Western Development Lab. Div.
Mailstop X-20
3939 Fabian Way
Palo Alto, CA 94303
(415) 852-4198
KLY@FORD-WDL1

* Christine Youngblut
Advanced Software Methods, Inc.
17021 Sioux Lane
Gaithersburg, MD 20878
(301) 948-1989
CYOUNGBLUT@ECLB

* Margie Zuk
Mailstop B321, Bldg B
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7590
MMZ@MITRE-BEDFORD

END

11-86

DTIC